EPTCS 392

Proceedings of the

First Workshop on Trends in Configurable Systems Analysis

Paris, France, 23rd April 2023

Edited by: Maurice H. ter Beek and Clemens Dubslaff

Published: 31st October 2023
DOI: 10.4204/EPTCS.392
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

Maurice H. ter Beek
Formal Methods and Tools (FMT) lab
Institute of Information Science and Technologies (ISTI)
National Research Council (CNR)
Pisa, Italy
maurice.terbeek@isti.cnr.it

Clemens Dubslaff
Formal System Analysis (FSA) group
Department of Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
c.dubslaff@tue.nl

These post-proceedings contain a selection of papers presented at the *First Workshop on Trends in Configurable Systems Analysis (TiCSA)*, which took place on 23 April 2023 in Paris, France, as a satellite event of the 26th European Joint Conferences on Theory And Practice of Software (ETAPS 2023).

TiCSA's primary goal was to bring together researchers active in the areas of design and analysis of configurable systems, discuss trends and novel analysis methods, and foster collaboration. Co-locating the workshop with ETAPS provided an excellent opportunity for exchanging results and experiences on applying analysis techniques from formal methods and software engineering to configurable systems.

System variants often arise by configuring parameters that have a direct impact on the system's behavior. Most prominently, in feature-oriented system design, features describe optional or incremental system functionalities whose configuration is simply whether a feature is active or inactive. Since the configuration space usually suffers from an exponential blowup in the number of configuration parameters, such *configurable systems* require specialized methods for their design and analysis. While there have been significant advances for the analysis of configurable systems in the last decade, most prominently *family-based analysis* or *feature-based analysis*, tackling the exponential blowup, there are still manifold opportunities that have not yet been considered. These cover specialized algorithms concerning *quantitative* and *compositional* aspects, required to model and analyze cyber-physical systems, as well as *adaptivity* and *reconfigurations* that allow for system adaptations in context-aware system design.

TiCSA is a continuation and extension of the QAVS series, which comprised three workshops with a focus on quantitative aspects in configurable systems analysis. These workshops had to be unfortunately held online only due to the Covid crisis, and no proceedings were published in the three editions. Hence, these proceedings can be seen also the first in the workshop series. The first QAVS workshop was co-located with QONFEST 2020 and comprised six presentations with around 20 attendees. Sven Apel (Saarland University, Germany) and Axel Legay (UC Louvain, Belgium) were invited to give keynotes. The second QAVS workshop in 2021 was the first under the umbrella of ETAPS with four presentations and about 15 attendees. Norbert Siegmund (University of Leipzig, Germany) was the invited speaker. The third QAVS workshop was co-located with ETAPS 2022 with five presentations and around 15 attendees. Jan Křetínský (TU Munich, Germany) provided an invited keynote. The decision to broaden the scope of the workshop, by considering other trends in configurable systems analysis rather than quantitative aspects alone, arose from audience feedback and discussions during the last QAVS workshop at ETAPS 2022.

TiCSA received six extended abstract submissions, which were accepted for a presentation after being reviewed for suitability. The workshop was opened by a keynote talk by Étienne André (Sorbonne Paris North University, France) on *Configuring Timing Parameters to Ensure Opacity*. Furthermore, an inspiration talk by Alfons Laarman (Leiden University, The Netherlands) on *LIMDD – A Decision Diagram for Simulation of Quantum Computing* initiated a discussion around quantum configurable systems and configurable quantum systems, concluding the workshop. We hereby thank Étienne and Alfons for accepting our invitations and all authors who submitted their work for their contributions. After the workshop, we received three full papers for publication in these post-proceedings, all of which were accepted after having been reviewed by no less than four members of the program committee.

We would like to thank the program committee members, listed below, for their careful and swift reviewing. We are also grateful to the ETAPS workshop chairs, Benedikt Bollig and Stefan Haar (Paris Saclay University, France), for accepting TiCSA as a satellite event at ETAPS 2023 and to the latter's general chairs Fabrice Kordon (Sorbonne University, France) and Laure Petrucci (Sorbonne Paris North University, France) and their team for the smooth organization and the pleasant interaction concerning organizational matters. We would also like to take this opportunity to thank EasyChair, which automates most of the tasks involved in organizing and chairing a workshop. Finally, we thank EPTCS and its editor-in-chief, Rob van Glabbeek, for agreeing to publish the proceedings of TiCSA 2023.

Program Chairs

Maurice ter Beek (ISTI–CNR, Pisa, Italy)

Clemens Dubslaff (Eindhoven University of Technology, The Netherlands)

Program Committee

Sven Apel (Saarland University, Germany)

Davide Basile (ISTI–CNR, Pisa, Italy)

Philipp Chrszon (German Aerospace Center, Braunschweig, Germany)

Erik de Vink (Eindhoven University of Technology, The Netherlands)

Uli Fahrenberg (LIX, Palaiseau, France)

Sebastian Junges (Radboud University Nijmegen, The Netherlands)

Jan Křetínský (Technical University of Munich, Germany)

Axel Legay (UC Louvain, Belgium)

Alberto Lluch Lafuente (Technical University of Denmark)

Tatjana Petrov (University of Kostanz, Germany)

José Proença (CISTER, Polytechnic Institute of Porto, Portugal)

Genaína Rodrigues (University of Brasília, Brazil)

Christoph Seidl (IT University of Copenhagen, Denmark)

Thomas Thüm (University of Ulm, Germany)

Andrea Vandin (Sant'Anna School of Advanced Studies, Pisa, Italy)

Mahsa Varshosaz (IT University, Copenhagen, Denmark)

Configuring Timing Parameters to Ensure Execution-Time Opacity in Timed Automata*

Étienne André
Université Sorbonne Paris Nord, LIPN, CNRS UMR 7030
F-93430 Villetaneuse, France

Engel Lefaucheux
Université de Lorraine, CNRS, Inria, LORIA
F-54000 Nancy, France

Didier Lime
Nantes Université, École Centrale Nantes, CNRS, LS2N, UMR 6004
F-44000 Nantes, France

Dylan Marinho
Université de Lorraine, CNRS, Inria, LORIA
F-54000 Nancy, France

Jun Sun
School of Computing and Information Systems
Singapore Management University

Timing information leakage occurs whenever an attacker successfully deduces confidential internal information by observing some timed information such as events with timestamps. Timed automata are an extension of finite-state automata with a set of clocks evolving linearly and that can be tested or reset, making this formalism able to reason on systems involving concurrency and timing constraints. In this paper, we summarize a recent line of works using timed automata as the input formalism, in which we assume that the attacker has access (only) to the system execution time. First, we address the following execution-time opacity problem: given a timed system modeled by a timed automaton, given a secret location and a final location, synthesize the execution times from the initial location to the final location for which one cannot deduce whether the secret location was visited. This means that for any such execution time, the system is opaque: either the final location is not reachable, or it is reachable with that execution time for both a run visiting and a run not visiting the secret location. We also address the full execution-time opacity problem, asking whether the system is opaque for all execution times; we also study a weak counterpart. Second, we add timing parameters, which are a way to configure a system: we identify a subclass of parametric timed automata with some decidability results. In addition, we devise a semi-algorithm for synthesizing timing parameter valuations guaranteeing that the resulting system is opaque. Third, we report on problems when the secret has itself an expiration date, thus defining expiring execution-time opacity problems. We finally show that our method can also apply to program analysis with configurable internal timings.

1 Introduction

Complex timed systems often combine hard real-time constraints with concurrency. Information leakage, notably through side channels (see, e.g., [23, 31]), can have dramatic consequences on the security of such systems. Among harmful information leaks, the *timing information leakage* (see, e.g., [22, 25, 38, 33, 35]) is the ability for an attacker to deduce internal information depending on observable timing information. In this paper, we focus on timing leakage through the total execution time, i.e., when a system works as an almost black-box and the ability of the attacker is limited to know the model and observe the total execution time. We consider here the formalism of timed automata (TAs) [1], which is a popular extension of finite-state automata with clocks measuring time, i.e., variables evolving linearly

*This work is partially supported by the ANR-NRF French-Singaporean research program ProMiS (ANR-19-CE25-0015 / 2019 ANR NRF 0092) and by ANR BisoUS (ANR-22-CE48-0012).

at the same rate. Such clocks can be tested against integer constants in locations ("invariants") or along transitions ("guards"), and can be reset to 0 when taking transitions.

Context and related works Franck Cassez proposed in [19] a first definition of *timed* opacity for TAs: the system is opaque if an attacker can never deduce whether some sequence of actions (possibly with timestamps) was performed, by only observing a given set of observable actions together with their timestamp. It is then proved in [19] that it is undecidable whether a TA is opaque, even for the restricted class of event-recording automata [2] (a subclass of TAs). This notably relates to the undecidability of timed language inclusion for TAs [1]. Security problems for TAs are surveyed in [13].

The aforementioned negative result leaves hope only if the definition or the setting is changed, which was done in three main lines of works. The different studied options were to reduce the expressiveness of the formalism [36, 37], to constrain the system to evolve in a time-bounded setting [4] or to consider a weaker attacker, who has access only to the *execution time* [9, 8], rather than to all observable actions with their timestamps. We present here a summary of our recent works in this latter setting [9, 8].

Contributions In the setting of TAs, we denote by *execution time* the time from the system start to the time a given (final) location is entered. Therefore, given a secret location, a TA is execution-time opaque (ET-opaque) for an execution time d if there exist at least two runs of duration d from the initial location to a final location: one visiting the secret location, and another one *not* visiting the secret location. In other words, if an attacker measures such an execution time from the initial location to the target location ℓ_f, then this attacker is not able to deduce whether the system visited ℓ_{priv}. Deciding whether at least one such d exists can be seen as an *existential* version of ET-opacity (called $\exists$-ET-opacity).

Then, a TA is *fully ET-opaque* if it is ET-opaque *for all execution times*: that is, for each possible execution time d, either the final location is unreachable, or the final location is reachable for at least two runs, one visiting the secret location, and another one not visiting it. We define a *weak* version of ET-opacity by only requiring that runs visiting the secret location on the way to the final location have a counterpart of the same duration not visiting the secret location on the way to the final location, but not necessarily the opposite: the TA is *weakly ET-opaque* if for each run visiting the secret location, there exists a run not visiting it with the same duration; the dual does not necessarily hold.

We also consider an *expiring version* of ET-opacity, where the secret is subject to an expiration date Δ. That is, we consider that an attack is successful only when the attacker can decide that the secret location was visited less than Δ time units before the system completion. Conversely, if the attacker exhibits an execution time d for which it is certain that the secret location was visited, but this location was visited strictly more than Δ time units prior to the system completion, then this attack is useless, and can be seen as a failed attack. The system is therefore *fully expiring ET-opaque* if the set of execution times for which the private location was visited within Δ time units prior to system completion (referred as "secret times") is exactly equal to the set of execution times for which the private location was either not visited or visited more than Δ time units prior to system completion (referred as "non-secret times"). Moreover, it is *weakly expiring ET-opaque* when the inclusion of the secret times into the non-secret ones is verified—and not necessarily the dual.

Finally, we study the aforementioned problems for a *parametric* extension of TAs, i.e., parametric timed automata (PTAs) [3], where integer constants compared to clocks can be made (rational-valued) timing parameters, i.e., unknown constants. Interesting problems include emptiness problems, i.e., the emptiness of the parameter valuations set such that (expiring) ET-opacity holds, and synthesis, i.e., the synthesis of all parameter valuations such that (expiring) ET-opacity holds.

Table 1: Summary of the results for ET-opacity [9]

		∃-ET-opaque	**weakly ET-opaque**	**fully ET-opaque**
Decision	TA	√(Proposition 2)	√**(Proposition 4)**	√(Proposition 3)
p-emptiness	L/U-PTA	√(Theorem 2)	×**(Theorem 6)**	×(Theorem 4)
	PTA	×(Theorem 1)	×**(Theorem 5)**	×(Theorem 3)
p-synthesis	L/U-PTA	×(Proposition 5)	×**(Corollary 5)**	×(Corollary 3)
	PTA	×(Corollary 1)	×**(Corollary 4)**	×(Corollary 2)

Table 2: Summary of the results for exp-ET-opacity [8]

		∃-exp-ET-opaque	**weakly exp-ET-opaque**	**fully exp-ET-opaque**
Decision	TA	√**(Theorem 9)**	√(Theorem 8)	√(Theorem 8)
Δ-emptiness	TA	?	√(Corollary 6)	√(Theorem 11)
Δ-computation	TA	?	√(Theorem 10)	?
Δ-p-emptiness	L/U-PTA	?	×(Theorem 12)	×(Theorem 12)
	PTA	?	×(Theorem 13)	×(Theorem 13)
Δ-p-synthesis	L/U-PTA	?	×(Corollary 7)	×(Theorem 12)
	PTA	?	×(Corollary 8)	×(Corollary 8)

About this manuscript This manuscript mainly summarizes results from two recent works, providing unified notations and concept names for the sake of consistency:

1. defining and studying ET-opacity problems [9] in TAs (Section 3) and PTAs (Section 4); these notions from [9] are presented differently (including the problem names) in this paper for sake of consistency; and
2. defining and studying expiring execution-time opacity (exp-ET-opacity) problems [8] in both TAs and PTAs (Section 5).

In addition, we prove a few original results on weak ET-opacity (that were not addressed in [9] because we had not yet defined the concept of weak ET-opacity when writing [9]) and on exp-ET-opacity. These original results are Propositions 2 and 4 and Theorems 5, 6 and 9.

In Tables 1 and 2, we summarize the decidability results recalled in this paper for ET-opacity and exp-ET-opacity. We denote a problem with a green check if it is decidable, with a red cross if it is undecidable, and with a yellow question mark if it is open (or not considered in the aforementioned papers [9, 8]). We emphasize using a bold font the original results of this paper. The p-emptiness (resp. p-synthesis) problem asks for the synthesis (resp. for the non-existence) of a parameter valuation for which ET-opacity is enforced. The Δ-p-synthesis (resp. emptiness) problem asks for the synthesis (resp. for the non-existence) of a parameter valuation and an expiring bound Δ for which the exp-ET-opacity is enforced. L/U-PTA denote the lower-bound/upper-bound parametric timed automata [27] subclass of PTAs. These notions will be formally defined in the paper.

Outline Section 2 recalls the necessary preliminaries, notably (parametric) timed automata. Section 3 defines and reviews execution-time opacity problems in timed automata. Section 4 defines and reviews execution-time opacity problems in timed automata. Section 5 defines and reviews *expiring* execution-time opacity problems in (parametric) timed automata. Section 6 briefly reports on our existing implementation of some of the problems using the parametric timed model checker IMITATOR [6]. Section 7 concludes the paper and reports on perspectives.

2 Preliminaries

We denote by $\mathbb{N}, \mathbb{Z}, \mathbb{Q}_{\geq 0}, \mathbb{R}_{\geq 0}$ the sets of non-negative integers, integers, non-negative rationals and non-negative reals, respectively.

2.1 Clocks, parameters and constraints

Clocks are real-valued variables that all evolve over time at the same rate. Throughout this paper, we assume a set $\mathbb{X} = \{x_1, \dots, x_H\}$ of *clocks*. A *clock valuation* is a function $\mu : \mathbb{X} \to \mathbb{R}_{\geq 0}$, assigning a non-negative value to each clock. We write $\vec{0}$ for the clock valuation assigning 0 to all clocks. Given a constant $d \in \mathbb{R}_{\geq 0}$, $\mu + d$ denotes the valuation s.t. $(\mu + d)(x) = \mu(x) + d$, for all $x \in \mathbb{X}$.

A *(timing) parameter* is an unknown rational-valued constant of a model. Throughout this paper, we assume a set $\mathbb{P} = \{p_1, \dots, p_M\}$ of *parameters*. A *parameter valuation* v is a function $v : \mathbb{P} \to \mathbb{Q}_{\geq 0}$.

As often, we choose *real*-valued clocks and *rational*-valued parameters, because irrational constants render reachability undecidable in TAs [30] (see [5] for a survey on the impact of these domains in (P)TAs).

We assume $\bowtie \in \{<, \leq, =, \geq, >\}$. A constraint C is a conjunction of inequalities over $\mathbb{X} \cup \mathbb{P}$ of the form $x \bowtie \sum_{1 \leq i \leq M} \alpha_i p_i + d$, with $p_i \in \mathbb{P}$, and $\alpha_i, d \in \mathbb{Z}$. Given C, we write $\mu \models v(C)$ if the expression obtained by replacing each x with $\mu(x)$ and each p with $v(p)$ in C evaluates to true.

2.2 Timed automata

A TA is a finite-state automaton extended with a finite set of real-valued clocks. We also add to the standard definition of TAs a special private location, which will be used to define our subsequent opacity concepts.

Definition 1 (Timed automaton [1])**.** A TA $\mathcal{A}$ is a tuple $\mathcal{A} = (\Sigma, L, \ell_0, \ell_{priv}, \ell_f, \mathbb{X}, I, E)$, where:

1. Σ is a finite set of actions,
2. L is a finite set of locations,
3. $\ell_0 \in L$ is the initial location,
4. $\ell_{priv} \in L$ is a special private location,
5. $\ell_f \in L$ is the final location,
6. $\mathbb{X}$ is a finite set of clocks,
7. I is the invariant, assigning to every $\ell \in L$ a constraint $I(\ell)$ over $\mathbb{X}$ (called *invariant*),
8. E is a finite set of edges $e = (\ell, g, a, R, \ell')$ where $\ell, \ell' \in L$ are the source and target locations, $a \in \Sigma$, $R \subseteq \mathbb{X}$ is a set of clocks to be reset, and g is a constraint over $\mathbb{X}$ (called *guard*).

Example 1. In Fig. 1, we give an example of a TA with three locations ℓ_0, ℓ_1 and ℓ_2, three edges, three actions $\{a, b, c\}$, and one clock x. ℓ_0 is the initial location, ℓ_2 is the private location, while ℓ_1 is the final location. ℓ_0 has an invariant $x \leq 3$ and the edge from ℓ_0 to ℓ_2 has a guard $x \geq 1$.

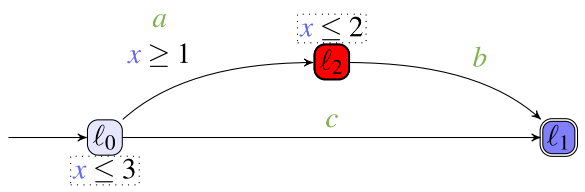


Figure 1: A TA example

Concrete semantics of timed automata We recall the concrete semantics of a TA using a timed transition system (TTS) [26].

Definition 2 (Semantics of a TA)**.** Given a TA $\mathscr{A} = (\Sigma, L, \ell_0, \ell_{priv}, \ell_f, \mathbb{X}, I, E)$, the semantics of $\mathscr{A}$ is given by the TTS $\mathfrak{T}_{\mathscr{A}} = (\mathfrak{S}, \mathfrak{s}_0, \Sigma \cup \mathbb{R}_{\geq 0}, \rightarrow)$, with

1. $\mathfrak{S} = \{(\ell, \mu) \in L \times \mathbb{R}^H_{\geq 0} \mid \mu \models I(\ell)v\}$,
2. $\mathfrak{s}_0 = (\ell_0, \vec{0})$,
3. $\rightarrow$ consists of the discrete and (continuous) delay transition relations:
   (a) discrete transitions: $(\ell, \mu) \overset{e}{\mapsto} (\ell', \mu')$, if $(\ell, \mu), (\ell', \mu') \in \mathfrak{S}$, and there exists $e = (\ell, g, a, R, \ell') \in E$, such that $\mu' = [\mu]_R$, and $\mu \models v(g)$.
   (b) delay transitions: $(\ell, \mu) \overset{d}{\mapsto} (\ell, \mu + d)$, with $d \in \mathbb{R}_{\geq 0}$, if $\forall d' \in [0, d], (\ell, \mu + d') \in \mathfrak{S}$.

Moreover we write $(\ell, \mu) \xrightarrow{(d,e)} (\ell', \mu')$ for a combination of a delay and discrete transition if $\exists \mu'' : (\ell, \mu) \overset{d}{\mapsto} (\ell, \mu'') \overset{e}{\mapsto} (\ell', \mu')$.

Given a TA $\mathscr{A}$ with concrete semantics $(\mathfrak{S}, \mathfrak{s}_0, \Sigma \cup \mathbb{R}_{\geq 0}, \rightarrow)$, we refer to the states of $\mathfrak{S}$ as the *concrete states* of $\mathscr{A}$. A *run* of $\mathscr{A}$ is an alternating sequence of concrete states of $\mathscr{A}$ and pairs of edges and delays starting from the initial state $\mathfrak{s}_0$ of the form $(\ell_0, \mu_0), (d_0, e_0), (\ell_1, \mu_1), \cdots$ with $i = 0, 1, \ldots, e_i \in E, d_i \in \mathbb{R}_{\geq 0}$ and $(\ell_i, \mu_i) \xrightarrow{(d_i, e_i)} (\ell_{i+1}, \mu_{i+1})$.

Definition 3 (Duration of a run)**.** Given a finite run $\rho : (\ell_0, \mu_0), (d_0, e_0), (\ell_1, \mu_1), \cdots, (d_{i-1}, e_{i-1}), (\ell_n, \mu_n)$, the *duration* of ρ is $dur(\rho) = \sum_{0 \leq i \leq n-1} d_i$. We also say that ℓ_n is reachable in time $dur(\rho)$.

Example 2. Consider again the TA $\mathscr{A}$ in Fig. 1. Consider the following run ρ of $\mathscr{A}$: $(\ell_0, x = 0), (1.4, a), (\ell_2, x = 1.4), (0.4, b), (\ell_1, x = 1.8)$ Note that we write "$x = 1.4$" instead of "μ such that $\mu(x) = 1.4$". We have $dur(\rho) = 1.4 + 0.4 = 1.8$.

2.3 Parametric timed automata

A PTA is a TA extended with a finite set of timing parameters allowing to model unknown constants.

Definition 4 (Parametric timed automaton [3])**.** A PTA $\mathscr{P}$ is a tuple $\mathscr{P} = (\Sigma, L, \ell_0, \ell_{priv}, \ell_f, \mathbb{X}, \mathbb{P}, I, E)$, where:

1. Σ is a finite set of actions;
2. L is a finite set of locations;
3. $\ell_0 \in L$ is the initial location;
4. $\ell_{priv} \in L$ is a special private location,
5. $\ell_f \in L$ is the final location;

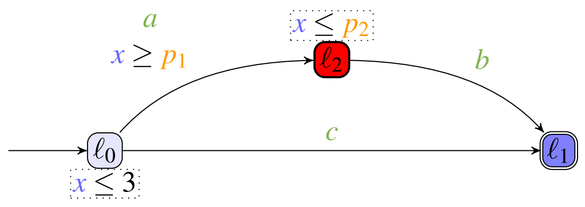


Figure 2: A PTA example

6. $\mathbb{X}$ is a finite set of clocks;
7. $\mathbb{P}$ is a finite set of parameters;
8. I is the invariant, assigning to every $\ell \in L$ a constraint $I(\ell)$ over $\mathbb{X} \cup \mathbb{P}$ (called *invariant*);
9. E is a finite set of edges $e = (\ell, g, a, R, \ell')$ where $\ell, \ell' \in L$ are the source and target locations, $a \in \Sigma$, $R \subseteq \mathbb{X}$ is a set of clocks to be reset, and g is a constraint over $\mathbb{X} \cup \mathbb{P}$ (called *guard*).

Example 3. In Fig. 2, we give an example of a PTA with three locations ℓ_0, ℓ_1 and ℓ_2, three edges, three actions $\{a,b,c\}$, one clock x and two parameters $\{p_1, p_2\}$. ℓ_0 is the initial location, ℓ_2 is the private location, while ℓ_1 is the final location. ℓ_0 has an invariant $x \leq 3$ and the edge from ℓ_0 to ℓ_2 has a guard $x \geq p_1$.

Definition 5 (Valuation of a PTA)**.** Given a parameter valuation v, we denote by $v(\mathscr{P})$ the non-parametric structure where all occurrences of a parameter p_i have been replaced by $v(p_i)$.

Remark 1. We have a direct correspondence between the valuation of a PTA and the definition of a TA given in Definition 1. TAs were originally defined with integer constants in [1] (as done in Definition 1), while our definition of PTAs allows *rational*-valued constants. By assuming a rescaling of the constants (i.e., by multiplying all constants in a TA by the least common multiple of their denominators), we obtain an equivalent (integer-valued) TA, as defined in Definition 1. So we assume in the following that $v(\mathscr{P})$ is a TA.

Example 4. Consider again the PTA in Fig. 2 and let v be such that $v(p_1) = 1$ and $v(p_2) = 2$. Then $v(\mathscr{P})$ is the TA depicted in Fig. 1.

Lower/upper parametric timed automaton While most decision problems are undecidable for the general class of PTAs (see [5] for a survey), lower/upper parametric timed automata (L/U-PTAs) [27] is the most well-known subclass of PTAs with some decidability results: for example, reachability-emptiness ("the emptiness of the valuations set for which a given location is reachable"), which is undecidable for PTAs [3], becomes decidable for L/U-PTAs [27]. Various other results were studied for this subclass (e.g., [17, 28, 12]).

Definition 6 (Lower/upper parametric timed automaton [27])**.** An L/U-PTA is a PTA where the set of parameters is partitioned into lower-bound parameters and upper-bound parameters, where each upper-bound (resp. lower-bound) parameter p_i must be such that, for every guard or invariant constraint $x \bowtie \sum_{1 \leq i \leq M} \alpha_i p_i + d$, we have:

- $\bowtie \in \{\leq, <\}$ implies $\alpha_i \geq 0$ (resp. $\alpha_i \leq 0$), and
- $\bowtie \in \{\geq, >\}$ implies $\alpha_i \leq 0$ (resp. $\alpha_i \geq 0$).

Example 5. The PTA in Fig. 2 is an L/U-PTA with $\{p_1\}$ as lower-bound parameter set, and $\{p_2\}$ as upper-bound parameter set.

3 Execution-time opacity problems in timed automata

Throughout this paper, the attacker model is as follows: the attacker knows the TA modeling the system, and can only observe the execution time between the start of the system and the time it reaches the final location. The attacker cannot observe actions, nor the values of the clocks, nor whether some locations are visited. Its goal will be to deduce from its observations whether the private location was visited.

3.1 Defining the execution times

Let us first introduce two key concepts necessary to define our notion of execution-time opacity.

Given a TA $\mathcal{A}$ and a run ρ, we say that ℓ_{priv} is *visited on the way to* ℓ_f *in* ρ if ρ is of the form

$$(\ell_0, \mu_0), (d_0, e_0), (\ell_1, \mu_1), \cdots, (\ell_m, \mu_m), (d_m, e_m), \cdots (\ell_n, \mu_n)$$

for some $m, n \in \mathbb{N}$ such that $\ell_m = \ell_{priv}$, $\ell_n = \ell_f$ and $\forall 0 \leq i \leq n-1, \ell_i \neq \ell_f$. We denote by $Visit^{priv}(\mathcal{A})$ the set of those runs, and refer to them as *private* runs. We denote by $DVisit^{priv}(\mathcal{A})$ the set of all the durations of these runs.

Conversely, we say that ℓ_{priv} is *avoided on the way to* ℓ_f *in* ρ if ρ is of the form

$$(\ell_0, \mu_0), (d_0, e_0), (\ell_1, \mu_1), \cdots, (\ell_n, \mu_n)$$

with $\ell_n = \ell_f$ and $\forall 0 \leq i < n, \ell_i \notin \{\ell_{priv}, \ell_f\}$. We denote the set of those runs by $\overline{Visit}^{priv}(\mathcal{A})$, referring to them as *public* runs, and by $D\overline{Visit}^{priv}(\mathcal{A})$ the set of all the durations of these public runs.

Therefore, $DVisit^{priv}(\mathcal{A})$ (resp. $D\overline{Visit}^{priv}(\mathcal{A})$) is the set of all the durations of the runs for which ℓ_{priv} is visited (resp. avoided) on the way to ℓ_f.

These concepts can be seen as the set of execution times from the initial location ℓ_0 to the final location ℓ_f while visiting (resp. not visiting) a private location ℓ_{priv}. Observe that, from the definition of the duration of a run (Definition 3), this "execution time" does not include the time spent in ℓ_f.

Example 6. Consider again the TA in Fig. 1. We have $DVisit^{priv}(\mathcal{A}) = [1,2]$ and $D\overline{Visit}^{priv}(\mathcal{A}) = [0,3]$.

3.2 Defining execution-time opacity

We now introduce formally the concept of "ET-opacity for a set of durations (or execution times) D": a system is *ET-opaque for execution times* D whenever, for any duration in D, it is not possible to deduce whether the system visited ℓ_{priv} or not. In other words, if an attacker measures an execution time within D from the initial location to the target location ℓ_f, then this attacker is not able to deduce whether the system visited ℓ_{priv}.

Definition 7 (Execution-time opacity (ET-opacity) for D)**.** Given a TA $\mathcal{A}$ and a set of execution times D, we say that $\mathcal{A}$ is *execution-time opaque (ET-opaque) for execution times* D if $D \subseteq (DVisit^{priv}(\mathcal{A}) \cap D\overline{Visit}^{priv}(\mathcal{A}))$.

In the following, we will be interested in the existence of such an execution time. We say that a TA is $\exists$-ET-opaque if it is ET-opaque for a non-empty set of execution times.

Definition 8 ($\exists$-ET-opacity)**.** A TA $\mathcal{A}$ is $\exists$-*ET-opaque* if $(DVisit^{priv}(\mathcal{A}) \cap D\overline{Visit}^{priv}(\mathcal{A})) \neq \emptyset$.

If one does not have the ability to tune the system (i.e., change internal delays, or add some `Thread.sleep()` statements in a program), one may be first interested in knowing whether the system is ET-opaque for all execution times. In other words, if a system is *fully ET-opaque*, for any possible measured execution time, an attacker is not able to deduce whether ℓ_{priv} was visited or not.

Definition 9 (full ET-opacity)**.** A TA $\mathscr{A}$ is *fully ET-opaque* if $\mathit{DVisit}^{priv}(\mathscr{A}) = \mathit{D}\overline{\mathit{Visit}}^{priv}(\mathscr{A})$.

That is, a system is fully ET-opaque if, for any execution time d, a run of duration d reaches ℓ_f after visiting ℓ_{priv} iff another run of duration d reaches ℓ_f without visiting ℓ_{priv}.

Remark 2. This definition is symmetric: a system is not fully ET-opaque iff an attacker can deduce ℓ_{priv} or $\neg\ell_{priv}$. For instance, if there is no run to ℓ_f visiting ℓ_{priv}, but still a run to ℓ_f (not visiting ℓ_{priv}), a system is not fully ET-opaque w.r.t. Definition 9.

We finally define weak ET-opacity, not considered in [9], but defined in the specific context of expiring opacity [8]. We therefore reintroduce this definition in the "normal" opacity setting considered in this section, in the following:

Definition 10 (weak ET-opacity)**.** A TA $\mathscr{A}$ is *weakly ET-opaque* if $\mathit{DVisit}^{priv}(\mathscr{A}) \subseteq \mathit{D}\overline{\mathit{Visit}}^{priv}(\mathscr{A})$.

That is, a TA is weakly ET-opaque whenever, for any run reaching the final location after visiting the private location, there exists another run of the same duration reaching the final location but not visiting the private location; but the converse does not necessarily hold.

Remark 3. Our notion of weak ET-opacity may still leak some information: on the one hand, if a run indeed visits the private location, there exists an equivalent run not visiting it, and therefore the system is ET-opaque; *but* on the other hand, there may exist execution times for which the attacker can deduce that the private location was *not* visited. This remains acceptable in some cases, and this motivates us to define a weak version of ET-opacity. Also note that the "initial-state opacity" for real-time automata considered in [36] can also be seen as *weak* in the sense that their language inclusion is also unidirectional.

Example 7. Consider again the PTA $\mathscr{P}$ in Fig. 2 and let v such that $v(p_1) = 1$ while $v(p_2) = 2$ (i.e., the TA in Fig. 1). Recall that $\mathit{DVisit}^{priv}(v(\mathscr{P})) = [1,2]$ and $\mathit{D}\overline{\mathit{Visit}}^{priv}(v(\mathscr{P})) = [0,3]$. Hence, it holds that $\mathit{DVisit}^{priv}(v(\mathscr{P})) \subseteq \mathit{D}\overline{\mathit{Visit}}^{priv}(v(\mathscr{P}))$ and therefore $v(\mathscr{P})$ is weakly ET-opaque. However, $\mathit{DVisit}^{priv}(v(\mathscr{P})) \neq \mathit{D}\overline{\mathit{Visit}}^{priv}(v(\mathscr{P}))$ and therefore $v(\mathscr{P})$ is not fully ET-opaque.

Now consider again the PTA $\mathscr{P}$ in Fig. 2 and let v' such that $v'(p_1) = 0$ while $v'(p_2) = 3$. This time, $\mathit{DVisit}^{priv}(v'(\mathscr{P})) = \mathit{D}\overline{\mathit{Visit}}^{priv}(v'(\mathscr{P})) = [0,3]$ and therefore $v'(\mathscr{P})$ is fully ET-opaque.

3.3 Decision and computation problems

3.3.1 Computation problem for ET-opacity

We can now define the ET-opacity t-computation problem, which consists in computing the possible execution times ensuring ET-opacity.

ET-opacity t-computation problem:
INPUT: A TA $\mathscr{A}$
PROBLEM: Compute the execution times D such that $\mathscr{A}$ is ET-opaque for D.

Let us illustrate that this computation problem is certainly not easy. For the TA $\mathscr{A}$ in Fig. 3, the execution times D for which $\mathscr{A}$ is ET-opaque is exactly $\mathbb{N}$; that is, only integer times ensure ET-opacity (as the system can only leave ℓ_{priv} and hence enter ℓ_f at an integer time), while non-integer times violate ET-opacity.

3.3.2 Decision problems

We define the three following decision problems:

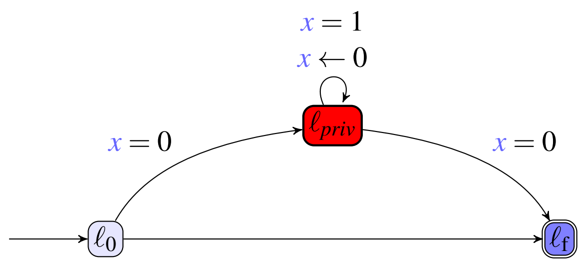


Figure 3: TA for which the set of execution times ensuring ET-opacity is $\mathbb{N}$

∃-ET-opacity decision problem:
INPUT: A TA $\mathcal{A}$
PROBLEM: Is $\mathcal{A}$ ∃-ET-opaque?

Full ET-opacity decision problem:
INPUT: A TA $\mathcal{A}$
PROBLEM: Is $\mathcal{A}$ fully ET-opaque?

Weak ET-opacity decision problem:
INPUT: A TA $\mathcal{A}$
PROBLEM: Is $\mathcal{A}$ weakly ET-opaque?

3.4 Answering the ET-opacity t-computation problem

Proposition 1 (Solvability of the ET-opacity t-computation problem [9, Proposition 5.2])**.** *The ET-opacity t-computation problem is solvable for TAs.*

This positive result can be put in perspective with the negative result of [19] that proves that it is undecidable whether a TA (and even the more restricted subclass of event-recording automata (ERAs) [2]) is opaque, in a sense that the attacker can deduce some actions, by looking at observable actions together with their timing. The difference in our setting is that only the global time is observable, which can be seen as a single action, occurring once only at the end of the computation. In other words, our attacker is less powerful than the attacker in [19].

3.5 Checking for ∃-ET-opacity

The following result was not strictly speaking proved in [9], and we provide here an original proof for it.

Proposition 2 (Decidability of the ∃-ET-opacity decision problem)**.** *The ∃-ET-opacity decision problem is decidable in* 5EXPTIME *for TAs.*

Proof. Let $\mathcal{A}$ be a TA. Suppose we add a Boolean variable *priv* to $\mathcal{A}$ which is initially false and set to true on every edge going into the location ℓ_{priv}. This Boolean variable (not strictly part of the TA syntax) can also be simulated by adding a copy of $\mathcal{A}$ instead, and jumping to that copy on edges going into location ℓ_{priv}.

Then the ∃-ET-opacity decision problem amounts to checking the following parametric TCTL formula [18], with p a parameter:

$$\exists p \big(\exists \Diamond_{=p} (\ell_f \land priv) \land \exists \Diamond_{=p} (\ell_f \land \neg priv) \big)$$

From [18], this can be checked in 5EXPTIME, since the size of the TA it is checked on is at most twice that of $\mathscr{A}$, and the size of the formula is constant w.r.t. the size of $\mathscr{A}$. □

3.6 Checking for full ET-opacity

The following result matches [9, Proposition 5.3] but we provide an original proof, also fixing a complexity issue in [9, Proposition 5.3].

Proposition 3 (Decidability of the full ET-opacity decision problem)**.** *The full ET-opacity decision problem is decidable in* 5EXPTIME *for TAs.*

Proof. As before, we can write a parametric TCTL formula for this problem, with p a parameter:

$$\forall p \big(\exists \Diamond_{=p} (\ell_{\mathrm{f}} \wedge priv) \Leftrightarrow \exists \Diamond_{=p} (\ell_{\mathrm{f}} \wedge \neg priv) \big)$$

This formula can be checked in 5EXPTIME [18]. □

3.7 Checking for weak ET-opacity

The *weak* notion of ET-opacity had not been defined in [9]. Nevertheless, the proof of Proposition 3 can be adapted in a very straightforward manner to prove its weak counterpart as follows:

Proposition 4 (Decidability of the weak ET-opacity decision problem)**.** *The weak ET-opacity decision problem is decidable in* 5EXPTIME *for TAs.*

Proof. Let $\mathscr{A}$ be a TA. As before, we can write a parametric TCTL formula for this problem, with p a parameter:

$$\forall p \big(\exists \Diamond_{=p} (\ell_{\mathrm{f}} \wedge priv) \Rightarrow \exists \Diamond_{=p} (\ell_{\mathrm{f}} \wedge \neg priv) \big)$$

This formula can be checked in 5EXPTIME [18]. □

4 Execution-time opacity problems in parametric timed automata

We now extend opacity problems to parametric timed automata. We first address the parametric problems related to $\exists$-ET-opacity in Section 4.1. The decision problems associated to full ET-opacity and weak ET-opacity will then be considered in Sections 4.2 and 4.3 respectively.

Following the usual concepts for parametric timed automata, we consider both *emptiness* and *synthesis* problems. An emptiness problem aims at *deciding* whether the set of parameter valuations for which a given property holds in the valuated TA is empty, while a synthesis problem aims at *synthesizing* the set of parameter valuations for which a given property holds in the valuated TA.

4.1 $\exists$-ET-opacity

4.1.1 Problems

Emptiness problem for $\exists$-ET-opacity Let us consider the following decision problem, i.e., the problem of checking the *emptiness* of the set of parameter valuations guaranteeing $\exists$-ET-opacity.

$\exists$-ET-opacity p-emptiness problem:
INPUT: A PTA $\mathscr{P}$
PROBLEM: Decide the emptiness of the set of parameter valuations v such that $v(\mathscr{P})$ is $\exists$-ET-opaque.

The negation of the $\exists$-ET-opacity p-emptiness problem consists in deciding whether there exists at least one parameter valuation for which $v(\mathscr{P})$ is $\exists$-ET-opaque.

Synthesis problem for $\exists$-ET-opacity The synthesis counterpart allows for a higher-level problem by also synthesizing the internal timings guaranteeing $\exists$-ET-opacity.

$\exists$-ET-opacity p-synthesis problem:
INPUT: A PTA $\mathscr{P}$
PROBLEM: Synthesize the set V of parameter valuations such that $v(\mathscr{P})$ is $\exists$-ET-opaque, for all $v \in V$.

4.1.2 Undecidability in general

With the rule of thumb that all non-trivial decision problems are undecidable for general PTAs [5], the following result is not surprising, and follows from the undecidability of reachability-emptiness for PTAs [3].

Theorem 1 (Undecidability of the $\exists$-ET-opacity p-emptiness problem [9, Theorem 6.1])**.** *The $\exists$-ET-opacity p-emptiness problem is undecidable for general PTAs.*

Since the emptiness problem is undecidable, the synthesis problem is immediately unsolvable as well.

Corollary 1. *The $\exists$-ET-opacity p-synthesis problem is unsolvable for general PTAs.*

Nevertheless, in [9] we proposed a procedure solving this problem. While this procedure is not guaranteed to terminate, its result is correct when termination can be achieved. See [9, Section 8] for details.

4.1.3 The subclass of L/U-PTAs

Decidability We now show that the $\exists$-ET-opacity p-emptiness problem is decidable for L/U-PTAs. Despite early positive results for L/U-PTAs [27, 17], more recent results (notably [28, 11, 12]) mostly proved undecidable properties of L/U-PTAs, and therefore this positive result is welcome.

Theorem 2 (Decidability of the $\exists$-ET-opacity p-emptiness problem [9, Theorem 6.2])**.** *The $\exists$-ET-opacity p-emptiness problem is decidable for L/U-PTAs.*

Intractability of synthesis for lower/upper parametric timed automata Even though the $\exists$-ET-opacity p-emptiness problem is decidable for L/U-PTAs (Theorem 2), the *synthesis* of the parameter valuations remains intractable in general, as shown in the following Proposition 5. By intractable we mean more precisely that the solution, if it can be computed, cannot (in general, i.e., for some sufficiently complex solutions) be represented using any formalism for which the emptiness of the intersection with equality constraints is decidable. That is, a formalism in which it is decidable to decide "the emptiness of the valuation set of the computed solution intersected with an equality test between variables" cannot be used to represent the solution. For example, let us question whether we could represent the solution of the $\exists$-ET-opacity p-synthesis problem for L/U-PTAs using the formalism of a *finite union of polyhedra*: testing whether a finite union of polyhedra intersected with "equality constraints" (typically $p_1 = p_2$) is empty or not *is* decidable. The Parma polyhedra library [14] can typically compute the answer to this question. Therefore, from the following Proposition 5, finite unions of polyhedra cannot be used to represent the solution of the $\exists$-ET-opacity p-synthesis problem for L/U-PTAs. As finite unions of polyhedra are a very common formalism (not to say the *de facto* standard) to represent the solutions of various timing parameters synthesis problems, the synthesis is then considered to be infeasible in practice, or *intractable* (following the vocabulary used in [28, Theorem 2]).

Proposition 5 (Intractability of the $\exists$-ET-opacity p-synthesis problem [9, Proposition 6.4])**.** *In case a solution to the $\exists$-ET-opacity p-synthesis problem for L/U-PTAs can be computed, this solution may be not representable using any formalism for which the emptiness of the intersection with equality constraints is decidable.*

4.2 Parametric full ET-opacity

We address here the following decision problem, which asks about the emptiness of the parameter valuation set guaranteeing full ET-opacity. We also define the full ET-opacity p-synthesis problem, this time *synthesizing* the timing parameters guaranteeing full ET-opacity.

4.2.1 Problem definitions

Full ET-opacity p-emptiness problem:
INPUT: A PTA $\mathscr{P}$
PROBLEM: Decide the emptiness of the set of parameter valuations v such that $v(\mathscr{P})$ is fully ET-opaque.

Equivalently, we are interested in deciding whether there exists at least one parameter valuation for which $v(\mathscr{P})$ is fully ET-opaque.

We also define the *full ET-opacity p-synthesis problem*, aiming at synthesizing (ideally the entire set of) parameter valuations v for which $v(\mathscr{P})$ is fully ET-opaque.

Full ET-opacity p-synthesis problem:
INPUT: A PTA $\mathscr{P}$
PROBLEM: Synthesize the set V of parameter valuations such that $v(\mathscr{P})$ is fully ET-opaque, for all $v \in V$.

4.2.2 Undecidability for general PTAs

Considering that Theorem 1 shows the undecidability of the $\exists$-ET-opacity p-emptiness problem, the undecidability of the full ET-opacity p-emptiness problem is not surprising, but does not follow immediately.

Theorem 3 (Undecidability of the full ET-opacity p-emptiness problem [9, Theorem 7.2])**.** *The full ET-opacity p-emptiness problem is undecidable for general PTAs.*

The proof relies on a reduction from the problem of reachability-emptiness in constant time, a result proved itself undecidable in the same paper [9, Lemma 7.1].

Since the emptiness problem is undecidable, the synthesis problem is immediately unsolvable as well.

Corollary 2. *The full ET-opacity p-synthesis problem is unsolvable for PTAs.*

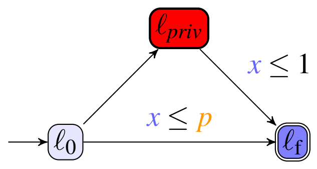


Figure 4: No monotonicity for full ET-opacity in L/U-PTAs

4.2.3 Undecidability for lower/upper parametric timed automata

Let us now study the full ET-opacity p-emptiness problem for L/U-PTAs. While it is well-known that L/U-PTAs enjoy a monotonicity for reachability properties ("enlarging an upper-bound parameter or decreasing a lower-bound parameter preserves reachability") [27], we can show in the following example that this is not the case for full ET-opacity.

Example 8. Consider the PTA in Fig. 4. First assume v such that $v(p) = 0.5$. Then, $v(\mathscr{P})$ is not fully ET-opaque: indeed, ℓ_f can be reached in 1 time unit by visiting ℓ_{priv}, but not without visiting ℓ_{priv}.

Second, assume v' such that $v'(p) = 1$. Then, $v'(\mathscr{P})$ is fully ET-opaque: indeed, ℓ_f can be reached for any duration in $[0, 1]$ by runs both visiting and not visiting ℓ_{priv}.

Finally, let us enlarge p further, and assume v'' such that $v''(p) = 2$. Then, $v''(\mathscr{P})$ becomes again not fully ET-opaque: indeed, ℓ_f can be reached in 2 time units without visiting ℓ_{priv}, but cannot be reached in 2 time units by visiting ℓ_{priv}.

As a side note, remark that this PTA is actually an upper-bound parametric timed automaton (U-PTA) [17], that is, monotonicity for this problem does not even hold for U-PTAs.

In fact, we show that, while the $\exists$-ET-opacity p-emptiness problem is decidable for L/U-PTAs (Theorem 2), the full ET-opacity p-emptiness problem becomes undecidable for this same class. This confirms (after previous works in [17, 28, 11, 12]) that L/U-PTAs stand at the frontier between decidability and undecidability.

Theorem 4 (Undecidability of the full ET-opacity p-emptiness problem for L/U-PTAs [9, Theorem 7.4])**.** *The full ET-opacity p-emptiness problem is undecidable for L/U-PTAs.*

Since the emptiness problem is undecidable, the synthesis problem is immediately unsolvable as well.

Corollary 3. *The full ET-opacity p-synthesis problem is unsolvable for L/U-PTAs.*

Remark 4. Since L/U-PTAs are a subclass of PTAs (put it differently: "any L/U-PTA is a PTA"), the negative results proved for L/U-PTAs (Theorem 4 and Corollary 3) immediately imply those previously shown for general PTAs (Theorem 3 and Corollary 2). However, in [9], a smaller number of clocks and parameters is needed to prove the aforementioned negative results for general PTAs, which justifies the two versions of the proofs in [9].

4.3 Parametric weak ET-opacity

4.3.1 Problem definitions

Weak ET-opacity p-emptiness problem:
INPUT: A PTA $\mathscr{P}$
PROBLEM: Decide the emptiness of the set of parameter valuations v such that $v(\mathscr{P})$ is weakly ET-opaque.

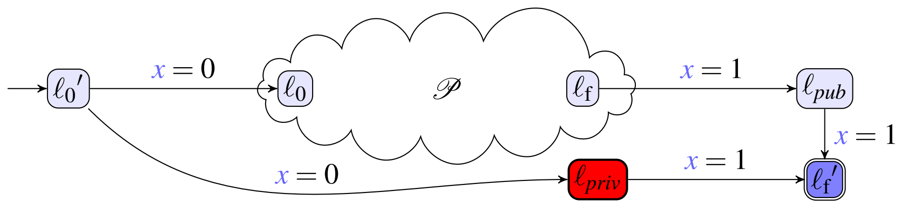


Figure 5: Reduction from reachability-emptiness for the proof of Theorem 5

Weak ET-opacity p-synthesis problem:
INPUT: A PTA $\mathscr{P}$
PROBLEM: Synthesize the parameter valuations v such that $v(\mathscr{P})$ is weakly ET-opaque.

4.3.2 Undecidability for general PTAs

We provide below an original result in the context of *weak* opacity, but partially inspired by the construction used in the proof of Theorem 3.

Theorem 5 (Undecidability of the weak ET-opacity p-emptiness problem)**.** *The weak ET-opacity p-emptiness problem is undecidable for general PTAs.*

Proof. We reduce from the reachability-emptiness problem in bounded time, which is undecidable from [10, Theorem 3.12]. (This is different from the proof of [9, Theorem 7.2], which reduces from the reachability-emptiness problem in *constant* time, which is undecidable according to [9, Lemma 7.1].)

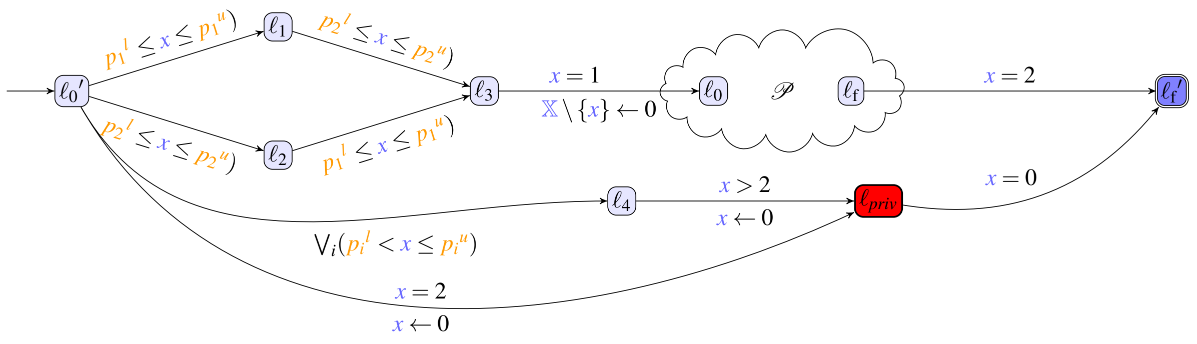
Consider an arbitrary PTA $\mathscr{P}$, with initial location ℓ_0 and a final location ℓ_f. We add the following locations and transitions in $\mathscr{P}$ to obtain a PTA $\mathscr{P}'$, as in Fig. 5: (*i*) a new initial location $\ell_0{}'$, with outgoing transitions in 0-time (due to their guard $x = 0$, where x is a new clock not belonging to $\mathscr{P}$, and never reset in $\mathscr{P}'$) to ℓ_0 and to a new location ℓ_{priv}, (*ii*) a new location ℓ_{pub} with an incoming transition from ℓ_f guarded by $x = 1$, and (*iii*) a new final location $\ell_f{}'$ with incoming transitions from ℓ_{pub} and ℓ_{priv} both guarded by $x = 1$.

First, note that, due to the guarded transitions, $\ell_f{}'$ is reachable for any parameter valuation via runs visiting ℓ_{priv}, (only) for an execution time equal to 1. That is, for all v, $\mathit{DVisit}^{priv}(v(\mathscr{P}')) = \{1\}$.

We now show that there exists a valuation v such that $v(\mathscr{P}')$ is weakly ET-opaque (with ℓ_{priv} as private location, and $\ell_f{}'$ as final location) iff there exists a valuation v such that ℓ_f is reachable in $v(\mathscr{P})$ for an execution time ≤ 1.

- $\Leftarrow$ Assume there exists some valuation v such that ℓ_f is reachable from ℓ_0 in $\mathscr{P}$ for an execution time ≤ 1. Then, due to our construction, ℓ_{pub} is visited on the way to $\ell_f{}'$ in $v(\mathscr{P}')$ (only) for the execution time 1. Therefore, $\overline{\mathit{DVisit}}^{priv}(v(\mathscr{P}')) = \{1\} = \mathit{DVisit}^{priv}(v(\mathscr{P}'))$ and then $v(\mathscr{P}')$ is weakly ET-opaque (and also fully ET-opaque, which plays no role here).

- $\Rightarrow$ Conversely, if ℓ_f is not reachable from ℓ_0 in $\mathscr{P}$ for any valuation for an execution time ≤ 1, then no run reaches $\ell_f{}'$ in time 1 without visiting ℓ_{priv}, for any valuation of $\mathscr{P}'$. Therefore, for any valuation v, $\mathit{DVisit}^{priv}(v(\mathscr{P}')) = \{1\} \not\subseteq \overline{\mathit{DVisit}}^{priv}(v(\mathscr{P}')) = \emptyset$. Therefore, there is no valuation v such that $v(\mathscr{P}')$ is weakly ET-opaque.



ℓ_0'
$p_1{}^l \leq x \leq p_1{}^u)$
ℓ_1
$p_2{}^l \leq x \leq p_2{}^u)$
$p_2{}^l \leq x \leq p_2{}^u)$
ℓ_2
$p_1{}^l \leq x \leq p_1{}^u)$
ℓ_3
$x = 1$
$\mathbb{X} \setminus \{x\} \leftarrow 0$
ℓ_0
$\mathscr{P}$
ℓ_f
$x = 2$
ℓ_f'
ℓ_4
$x > 2$
$x \leftarrow 0$
ℓ_{priv}
$x = 0$
$\bigvee_i (p_i{}^l < x \leq p_i{}^u)$
$x = 2$
$x \leftarrow 0$

Figure 6: Undecidability of full ET-opacity p-emptiness problem for L/U-PTAs

Therefore, there exists a valuation v such that $v(\mathscr{P}')$ is weakly ET-opaque iff there exists a valuation v such that ℓ_f is reachable in $v(\mathscr{P})$ for an execution time ≤ 1—which is undecidable from [10, Theorem 3.12]. This concludes the proof. $\square$

Since the emptiness problem is undecidable, the synthesis problem is immediately unsolvable as well.

Corollary 4. *The weak ET-opacity p-synthesis problem is unsolvable for general PTAs.*

4.3.3 Undecidability for lower/upper parametric timed automata

We provide below another original result in the context of *weak* opacity, this time for L/U-PTAs, largely inspired by the proof of Theorem 4, even though our construction needed to be changed.

Theorem 6 (Undecidability of the weak ET-opacity p-emptiness problem for L/U-PTAs)**.** *The weak ET-opacity p-emptiness problem is undecidable for L/U-PTAs.*

Proof. Let us recall from [10, Theorem 3.12] that the reachability-emptiness problem is undecidable over bounded time for PTAs with (at least) 3 clocks and 2 parameters. Assume a PTA $\mathscr{P}$ with 3 clocks and 2 parameters, say p_1 and p_2, and a final location ℓ_f. Take 1 as a time bound. From [10, Theorem 3.12], it is undecidable whether there exists a parameter valuation for which ℓ_f is reachable in $\mathscr{P}$ in time ≤ 1.

The idea of our proof is that, as in [28, 9], we "split" each of the two parameters used in $\mathscr{P}$ into a lower-bound parameter ($p_1{}^l$ and $p_2{}^l$) and an upper-bound parameter ($p_1{}^u$ and $p_2{}^u$). Each constraint of the form $x < p_i$ (resp. $x \leq p_i$) is replaced with $x < p_i{}^u$ (resp. $x \leq p_i{}^u$) while each constraint of the form $x > p_i$ (resp. $x \geq p_i$) is replaced with $x > p_i{}^l$ (resp. $x \geq p_i{}^l$); $x = p_i$ is replaced with $p_i{}^l \leq x \leq p_i{}^u$.

The idea is that the PTA $\mathscr{P}$ is exactly equivalent to our construction with duplicated parameters only when $p_1{}^l = p_1{}^u$ and $p_2{}^l = p_2{}^u$. The crux of the rest of this proof is that we will "rule out" any parameter valuation not satisfying these equalities, so as to use directly the undecidability result of [10, Theorem 3.12].

Now, consider the extension of $\mathscr{P}$ given in Fig. 6, and let $\mathscr{P}'$ be this extension. We assume that x is an extra clock not used in $\mathscr{P}$. The syntax "$\mathbb{X} \setminus \{x\} \leftarrow 0$" denotes that all clocks of the original PTA $\mathscr{P}$ are reset—but not the new clock x. The guard on the transition from ℓ_0' to ℓ_4 stands for 2 different transitions guarded with $p_1{}^l < x \leq p_1{}^u$, and $p_2{}^l < x \leq p_2{}^u$, respectively.

Let us first make the following observations:

1. for any parameter valuation, one can take the transition from ${\ell_0}'$ to ℓ_{priv} at time 2 and then to ${\ell_f}'$ in 0-time (i.e., at time 2), i.e., ${\ell_f}'$ is always reachable in time 2 while visiting location ℓ_{priv}; put differently, $\{2\} \subseteq \mathit{DVisit}^{priv}(v(\mathcal{P}'))$ for any parameter valuation v;

2. the original automaton $\mathcal{P}$ can only be entered whenever ${p_1}^l \leq {p_1}^u$ and ${p_2}^l \leq {p_2}^u$; going from ${\ell_0}'$ to ℓ_0 takes exactly 1 time unit (due to the $x = 1$ guard);

3. to reach ${\ell_f}'$ without visiting ℓ_{priv}, a run must go through $\mathcal{P}$ and visit ℓ_f, and its duration is necessarily 2; put differently, $\mathit{D\overline{Visit}}^{priv}(v(\mathcal{P}')) \subseteq \{2\}$ for any parameter valuation v;

4. from [10, Theorem 3.12], it is undecidable whether there exists a parameter valuation for which there exists a run reaching ℓ_f from ℓ_0 in time ≤ 1, i.e., reaching ℓ_f from ${\ell_0}'$ in time ≤ 2.

Let us consider the following cases depending on the valuations:

1. for valuations v such that ${p_1}^l > {p_1}^u$ or ${p_2}^l > {p_2}^u$, then thanks to the transitions from ${\ell_0}'$ to ℓ_0, there is no way to enter the original PTA $\mathcal{P}$ (and therefore to reach ${\ell_f}'$ without visiting ℓ_{priv}); hence, $\mathit{D\overline{Visit}}^{priv}(v(\mathcal{P}')) = \emptyset$, and therefore $\{2\} \subseteq \mathit{DVisit}^{priv}(v(\mathcal{P}')) \not\subseteq \mathit{D\overline{Visit}}^{priv}(v(\mathcal{P}'))$, i.e., $\mathcal{P}'$ is not weakly ET-opaque for any of these valuations.

2. for valuations v such that ${p_1}^l < {p_1}^u$ or ${p_2}^l < {p_2}^u$, then the transition from ${\ell_0}'$ to ℓ_4 can be taken, and therefore there exist runs reaching ${\ell_f}'$ after a duration > 2 (for example of duration 3) and visiting ℓ_{priv}. Since no run can reach ${\ell_f}'$ without visiting ℓ_{priv} for a duration $\neq 2$, then $\{3\} \subseteq \mathit{DVisit}^{priv}(v(\mathcal{P}')) \not\subseteq \mathit{D\overline{Visit}}^{priv}(v(\mathcal{P}')) \subseteq \{2\}$ and again $\mathcal{P}'$ is not weakly ET-opaque for any of these valuations.

3. for valuations such that ${p_1}^l = {p_1}^u$ and ${p_2}^l = {p_2}^u$, then the behavior of the modified $\mathcal{P}$ (with duplicate parameters) is exactly the one of the original $\mathcal{P}$. Also, note that the transition from ${\ell_0}'$ to ℓ_4 cannot be taken. In contrast, the transition from ${\ell_0}'$ to ℓ_{priv} can still be taken, and therefore there exists a run of duration 2 visiting ℓ_{priv} and reaching ${\ell_f}'$. Hence, $\mathit{DVisit}^{priv}(v(\mathcal{P}')) = \{2\}$ for any such valuation v.

   - Now, assume there exists such a parameter valuation v for which there exists a run of $v(\mathcal{P})$ of duration ≤ 1 reaching ℓ_f. And, as a consequence, there exists a run of $v(\mathcal{P}')$ of duration 2 (including the 1 time unit to go from ${\ell_0}'$ to ℓ_0) reaching ${\ell_f}'$ without visiting ℓ_{priv}. Hence, $\mathit{D\overline{Visit}}^{priv}(v(\mathcal{P}')) = \{2\}$. Therefore $\mathit{D\overline{Visit}}^{priv}(v(\mathcal{P}')) = \mathit{DVisit}^{priv}(v(\mathcal{P}')) = \{2\}$.

     As a consequence, the modified automaton $\mathcal{P}'$ is weakly ET-opaque (and actually fully ET-opaque—which plays no role in this proof) for such a parameter valuation.

   - Conversely, assume there exists no parameter valuation for which there exists a run of $\mathcal{P}$ of duration ≤ 1 reaching ℓ_f. In that case, ${\ell_f}'$ can never be reached without visiting ℓ_{priv}: $\mathit{D\overline{Visit}}^{priv}(v(\mathcal{P}')) = \emptyset$, and therefore $\{2\} \subseteq \mathit{DVisit}^{priv}(v(\mathcal{P}')) \not\subseteq \mathit{D\overline{Visit}}^{priv}(v(\mathcal{P}'))$, i.e., $v(\mathcal{P}')$ is not fully ET-opaque for any such parameter valuation v.

As a consequence, there exists a parameter valuation v' for which $v'(\mathcal{P}')$ is weakly ET-opaque iff there exists a parameter valuation v for which there exists a run in $v(\mathcal{P})$ of duration ≤ 1 reaching ℓ_f—which is undecidable from [10, Theorem 3.12]. □

Corollary 5. *The weak ET-opacity p-synthesis problem is unsolvable for L/U-PTAs.*

Table 3: Summary of the definitions for ET-opacity and expiring ET-opacity [9, 8]

	Secret runs	**Non-secret runs**
ET-opacity	Runs visiting the private location (= private runs)	Runs not visiting the private location (= public runs)
exp-ET-opacity	Runs visiting the private location $\leq \Delta$ time units before the system completion	(i) Runs not visiting the private location and (ii) Runs visiting the private location $> \Delta$ time units before the system completion

The system is **(resp. expiring)**	**if**
ET-opaque	$\{\text{secret runs}\} \cap \{\text{non-secret runs}\} \neq \emptyset$
weakly ET-opaque	$\{\text{secret runs}\} \subseteq \{\text{non-secret runs}\}$
full ET-opacity	$\{\text{secret runs}\} = \{\text{non-secret runs}\}$

5 Expiring execution-time opacity problems

In [4], the authors consider a time-bounded notion of the opacity of [19], where the attacker has to disclose the secret before an upper bound, using a partial observability. This can be seen as a secrecy with an *expiration date*. The rationale is that retrieving a secret "too late" is useless; this is understandable, e.g., when the secret depends of the status of the memory; if the cache was overwritten since, then knowing the secret is probably useless in most situations. In addition, the analysis in [4] is carried over a time-bounded horizon; this means there are two time bounds in [4]: one for the secret expiration date, and one for the bounded-time execution of the system.

In this section, we review a recent work of ours [8] in which we incorporate this secret expiration date into our notion of ET-opacity: we only consider the former notion of time bound from [4] (the secret expiration date), and lift the assumption regarding the latter (the bounded-time execution of the system). More precisely, we consider an *expiring version of ET-opacity*, where the secret is subject to an expiration date; this can be seen as a combination of both concepts from [9] and [4]. That is, we consider that an attack is successful only when the attacker can decide that the secret location was entered less than Δ time units before the system completion. Conversely, if the attacker exhibits an execution time d for which it is certain that the secret location was visited, but this location was entered strictly more than Δ time units prior to the system completion, then this attack is useless, and can be seen as a failed attack. The system is therefore *fully* exp-ET-opaque if the set of execution times for which the private location was entered within Δ time units prior to system completion is exactly equal to the set of execution times for which the private location was either not visited or entered $> \Delta$ time units prior to system completion.

In addition, when the former (secret) set of execution times is *included* into the latter (non-secret) set of times, we say that the system is *weakly* exp-ET-opaque; this encodes situations when the attacker might be able to deduce that no secret location was visited, but is not able to confirm that the secret location *was* indeed visited.

On the one hand, our attacker model is *less powerful* than [4], because our attacker has only access to the execution time (and to the input model); in that sense, our attacker capability is identical to [9]. On the other hand, we lift the time-bounded horizon analysis from [4], allowing to analyze systems without any assumption on their execution time; therefore, we only import from [4] the notion of *expiring secret*.

We summarize in Table 3 our different notions of ET-opacity and expiring ET-opacity; we will define formally expiring ET-opacity in the following.

5.1 Exp-ET-opacity

Let us first introduce some notions dedicated to expiring ET-opacity (hereafter referred to as exp-ET-opacity). Let $\mathbb{R}_{\geq 0}^{\infty} = \mathbb{R}_{\geq 0} \cup \{+\infty\}$. Given a TA $\mathscr{A}$ and a finite run ρ in $\mathfrak{T}_{\mathscr{A}}$, the *duration* between two states of $\rho : \mathfrak{s}_0, (d_0, e_0), \mathfrak{s}_1, \cdots, \mathfrak{s}_k$ is $dur_\rho(\mathfrak{s}_i, \mathfrak{s}_j) = \sum_{i \leq m \leq j-1} d_m$. We also define the *duration* between two locations ℓ_1 and ℓ_2 as the duration $dur_\rho(\ell_1, \ell_2) = dur_\rho(\mathfrak{s}_i, \mathfrak{s}_j)$ with $\rho : \mathfrak{s}_0, (d_0, e_0), \mathfrak{s}_1, \cdots, \mathfrak{s}_i, \cdots, \mathfrak{s}_j, \cdots, \mathfrak{s}_k$ where $\mathfrak{s}_j$ the first occurrence of a state with location ℓ_2 and $\mathfrak{s}_i$ is the last state of ρ with location ℓ_1 before $\mathfrak{s}_j$. We choose this definition to coincide with the definitions of opacity that we will define in the following Definition 11. Indeed, we want to make sure that revealing a secret (ℓ_1 in this definition) is not a failure if it is done after a given time. Thus, as soon as the system reaches its final state (ℓ_2), we will be interested in knowing how long the secret has been present, and thus the last time it was entered ($\mathfrak{s}_i$).

Given $\Delta \in \mathbb{R}_{\geq 0}^{\infty}$, we define $Visit_{\leq \Delta}^{priv}(\mathscr{A})$ (resp. $Visit_{> \Delta}^{priv}(\mathscr{A})$) as the set of runs $\rho \in Visit^{priv}(\mathscr{A})$ s.t. $dur_\rho(\ell_{priv}, \ell_f) \leq \Delta$ (resp. $dur_\rho(\ell_{priv}, \ell_f) > \Delta$). We refer to the runs of $Visit_{\leq \Delta}^{priv}(\mathscr{A})$ as *secret* runs; their durations are denoted by $DVisit_{\leq \Delta}^{priv}(\mathscr{A})$. Similarly, the durations of the runs of $Visit_{> \Delta}^{priv}(\mathscr{A})$ are denoted by $DVisit_{> \Delta}^{priv}(\mathscr{A})$.

We define below two notions of ET-opacity w.r.t. a time bound Δ. We will compare two sets:

1. the set of execution times for which the private location was entered at most Δ time units prior to system completion; and
2. the set of execution times for which either the private location was not visited at all, or it was last entered more than Δ time units prior to system completion (which, in our setting, is somehow similar to *not* visiting the private location, in the sense that entering it "too early" is considered of little interest).

If both sets match, the system is fully ($\leq \Delta$)-ET-opaque. If the former is included into the latter, then the system is weakly ($\leq \Delta$)-ET-opaque.

Definition 11 (Expiring Execution-time opacity)**.** Given a TA $\mathscr{A}$ and a bound (i.e., an expiration date for the secret) $\Delta \in \mathbb{R}_{\geq 0}^{\infty}$ we say that $\mathscr{A}$ is *fully exp-ET-opaque* w.r.t. the expiration date Δ, denoted by *fully ($\leq \Delta$)-ET-opaque*, if

$$DVisit_{\leq \Delta}^{priv}(\mathscr{A}) = DVisit_{> \Delta}^{priv}(\mathscr{A}) \cup D\overline{Visit}^{priv}(\mathscr{A}).$$

Moreover, $\mathscr{A}$ is *weakly exp-ET-opaque* w.r.t. the expiration date Δ, denoted by *weakly ($\leq \Delta$)-ET-opaque*, if

$$DVisit_{\leq \Delta}^{priv}(\mathscr{A}) \subseteq DVisit_{> \Delta}^{priv}(\mathscr{A}) \cup D\overline{Visit}^{priv}(\mathscr{A}).$$

Finally, $\mathscr{A}$ is $\exists$*-ET-opaque* w.r.t. the expiration date Δ, denoted by $\exists$*-($\leq \Delta$)-ET-opaque*, if

$$DVisit_{\leq \Delta}^{priv}(\mathscr{A}) \cap (DVisit_{> \Delta}^{priv}(\mathscr{A}) \cup D\overline{Visit}^{priv}(\mathscr{A})) \neq \emptyset.$$

Example 9. Consider again the PTA in Fig. 2; let v be such that $v(p_1) = 1$ and $v(p_2) = 2.5$. Fix $\Delta = 1$. We have:

- $D\overline{Visit}^{priv}(v(\mathscr{P})) = [0, 3]$
- $DVisit_{> \Delta}^{priv}(v(\mathscr{P})) = (2, 2.5]$
- $DVisit_{\leq \Delta}^{priv}(v(\mathscr{P})) = [1, 2.5]$

Therefore, we say that $v(\mathscr{P})$ is:

- $\exists$-(≤ 1)-ET-opaque, as $[1,2.5] \cap \big((2,2.5] \cup [0,3]\big) \neq \emptyset$
- weakly (≤ 1)-ET-opaque, as $[1,2.5] \subseteq \big((2,2.5] \cup [0,3]\big)$
- not fully (≤ 1)-ET-opaque, as $[1,2.5] \neq \big((2,2.5] \cup [0,3]\big)$

As noted in Remark 3, despite the weak ($\leq$ 1)-ET-opacity of $\mathscr{A}$, the attacker can deduce some information about the visit of the private location for some execution times. For example, if a run has a duration of 3 time units, it cannot be a private run, and therefore the attacker can deduce that the private location was not visited at all.

5.2 Exp-ET-opacity problems in timed automata

5.2.1 Problem definitions

We define seven different problems in the context of (non-parametric) TAs:

$\exists$-exp-ET-opacity decision problem:
INPUT: A TA $\mathscr{A}$ and a bound $\Delta \in \mathbb{R}^{\infty}_{\geq 0}$
PROBLEM: Decide whether $\mathscr{A}$ is $\exists$-($\leq \Delta$)-ET-opaque.

Full (resp. weak) exp-ET-opacity decision problem:
INPUT: A TA $\mathscr{A}$ and a bound $\Delta \in \mathbb{R}^{\infty}_{\geq 0}$
PROBLEM: Decide whether $\mathscr{A}$ is fully (resp. weakly) ($\leq \Delta$)-ET-opaque.

Full (resp. weak) exp-ET-opacity Δ-emptiness problem:
INPUT: A TA $\mathscr{A}$
PROBLEM: Decide the emptiness of the set of bounds Δ such that $\mathscr{A}$ is fully (resp. weakly) ($\leq \Delta$)-ET-opaque.

Full (resp. weak) exp-ET-opacity Δ-computation problem:
INPUT: A TA $\mathscr{A}$
PROBLEM: Compute the maximal set $\mathscr{D}$ of bounds such that $\mathscr{A}$ is fully (resp. weakly) ($\leq \Delta$)-ET-opaque for all $\Delta \in \mathscr{D}$.

Example 10. Consider again the PTA in Fig. 2; let v be such that $v(p_1) = 1$ and $v(p_2) = 2.5$ (as in Example 9). Let us exemplify some of the problems defined above.

- Given $\Delta = 1$, the weak exp-ET-opacity decision problem asks whether $v(\mathscr{P})$ is weakly ($\leq$ 1)-ET-opaque—the answer is "yes" from Example 9.
- The answer to the weak exp-ET-opacity Δ-emptiness problem is therefore "no" because the set of bounds Δ such that $v(\mathscr{P})$ is weakly ($\leq \Delta$)-ET-opaque is not empty.
- Finally, the weak exp-ET-opacity Δ-computation problem asks to compute all the corresponding bounds: in this example, the solution is $\Delta \in \mathbb{R}^{\infty}_{\geq 0}$, i.e., the solution is the set all possible (non-negative) values for Δ.

Relations with the ET-opacity problems Note that, when considering $\Delta = +\infty$, $DVisit^{priv}_{>\Delta}(\mathscr{A}) = \emptyset$ and all the execution times of runs visiting ℓ_{priv} are in $DVisit^{priv}_{\leq\Delta}(\mathscr{A})$. Therefore, full ($\leq +\infty$)-ET-opacity matches the full ET-opacity. We can therefore notice that answering the full exp-ET-opacity decision problem for $\Delta = +\infty$ is decidable (Proposition 3). However, the emptiness and computation problems cannot be reduced to full ET-opacity problems from Section 4.1.3.

Conversely, it is possible to answer the full ET-opacity decision problem by checking the full exp-ET-opacity decision problem with $\Delta = +\infty$. Moreover, the ET-opacity t-computation problem reduces to the full exp-ET-opacity Δ-computation problem: if $+\infty \in \mathscr{D}$, we get the answer.

Recall that we summarize our different definitions of (expiring) ET-opacity in Table 3.

5.2.2 Results

In general, the link between the full and weak notions of the three aforementioned problems is not obvious. However, for a fixed value of Δ, we establish the following theorem.

Theorem 7 ([8, Theorem 1])**.** *The full exp-ET-opacity decision problem reduces to the weak exp-ET-opacity decision problem.*

We can now study the aforementioned problems.

Theorem 8 (Decidability of full (resp. weak) exp-ET-opacity decision problem [8, Theorems 2 and 5])**.** *The full (resp. weak) exp-ET-opacity decision problem is decidable in* $\mathsf{NEXPTIME}$.

Remark 5. In Proposition 3, we established that the full ($\leq +\infty$)-ET-opacity decision problem is in $\mathsf{5EXPTIME}$. Theorem 8 thus extends our former results in three ways:

1. by including the parameter Δ,
2. by reducing the complexity and
3. by considering as well the *weak* notion of ET-opacity (considered separately in Proposition 4).

We complete these results from [8] with the following result analog to Proposition 2.

Theorem 9 (Decidability of $\exists$-exp-ET-opacity decision problem)**.** *The $\exists$-exp-ET-opacity decision problem is decidable in* PSPACE.

Proof. The full (resp. weak) exp-ET-opacity decision problem was solved in [8] by building two non-deterministic finite automata whose languages represented the secret and the non-secret durations of the system, respectively. These automata being of exponential size and with a unary language, testing the equality or inclusion of languages led to the $\mathsf{NEXPTIME}$ algorithm quoted in Theorem 8. Similarly, the $\exists$-exp-ET-opacity decision problem can be decided by testing whether the intersection of the languages of these automata is empty. This can be done in $\mathsf{NLOGSPACE}$ in the size of the automata (classically, by first building the product between these two automata, and then by checking the reachability of a pair of final states), hence the PSPACE algorithm. □

Theorem 10 (Solvability of weak exp-ET-opacity Δ-computation problem [8, Theorems 3 and 5])**.** *The weak exp-ET-opacity Δ-computation problem is solvable.*

Corollary 6 (Decidability of weak exp-ET-opacity Δ-emptiness problem [8, Corollary 1])**.** *The weak exp-ET-opacity Δ-emptiness problem is decidable.*

In contrast to the weak exp-ET-opacity Δ-computation problem, we only show below that the full exp-ET-opacity Δ-emptiness problem is decidable; the computation problem remains open.

Theorem 11 (Decidability of the full exp-ET-opacity Δ-emptiness problem [8, Theorems 4 and 5])**.** *The full exp-ET-opacity Δ-emptiness problem is decidable.*

5.3 Exp-ET-opacity in parametric timed automata

We now study exp-ET-opacity problems for PTAs: we will be interested in the synthesis and in the emptiness of the valuations set ensuring that a system is fully (resp. weakly) exp-ET-opaque.

5.3.1 Definitions

We define the following problems, where we ask for parameter valuations v and for valuations of Δ s.t. $v(\mathscr{P})$ is fully (resp. weakly) $(\leq \Delta)$-ET-opaque.

Full (resp. weak) exp-ET-opacity Δ-p-emptiness problem:
INPUT: A PTA $\mathscr{P}$
PROBLEM: Decide whether the set of parameter valuations v and valuations of Δ such that $v(\mathscr{P})$ is fully (resp. weakly) $(\leq \Delta)$-ET-opaque is empty

Full (resp. weak) exp-ET-opacity Δ-p-synthesis problem:
INPUT: A PTA $\mathscr{P}$
PROBLEM: Synthesize the set of parameter valuations v and valuations of Δ such that $v(\mathscr{P})$ is fully (resp. weakly) $(\leq \Delta)$-ET-opaque

Example 11. Consider again the PTA $\mathscr{P}$ in Fig. 2.

For this PTA, the answer to the weak exp-ET-opacity Δ-p-emptiness problem is false, as there exists such a valuation (e.g., the valuation given in Example 10).

Moreover, we can show that, for all Δ and v:

- $\overline{DVisit}^{priv}(v(\mathscr{P})) = [0,3]$
- if $v(p_1) > 3$ or $v(p_1) > v(p_2)$, it is not possible to reach ℓ_f with a run visiting ℓ_{priv} and therefore $DVisit^{priv}_{>\Delta}(v(\mathscr{P})) = DVisit^{priv}_{\leq\Delta}(v(\mathscr{P})) = \emptyset$
- if $v(p_1) \leq 3$ and $v(p_1) \leq v(p_2)$
  - $DVisit^{priv}_{>\Delta}(v(\mathscr{P})) = (v(p_1) + \Delta, v(p_2)]$
  - $DVisit^{priv}_{\leq\Delta}(v(\mathscr{P})) = [v(p_1), \min(\Delta + 3, v(p_2))]$

Recall that the full exp-ET-opacity Δ-p-synthesis problem aims at synthesizing the valuations such that $DVisit^{priv}_{\leq\Delta}(v(\mathscr{P})) = DVisit^{priv}_{>\Delta}(v(\mathscr{P})) \cup \overline{DVisit}^{priv}(v(\mathscr{P}))$. The answer to this problem is therefore the set of valuations of timing parameters and of Δ s.t.:

$$v(p_1) = 0 \land \Big(\big(\Delta \leq 3 \land 3 \leq v(p_2) \leq \Delta + 3\big) \lor \big(v(p_2) < \Delta \land v(p_2) = 3\big) \Big).$$

5.3.2 Results

The subclass of lower/upper parametric timed automata

Theorem 12 (Undecidability of full (resp. weak) exp-ET-opacity Δ-p-emptiness problem [8, Theorem 6])**.** *The full (resp. weak) exp-ET-opacity Δ-p-emptiness problem is undecidable for L/U-PTAs.*

The synthesis problems are therefore immediately unsolvable as well.

Corollary 7 ([8, Corollary 2])**.** *The full (resp. weak) exp-ET-opacity Δ-p-synthesis problem is unsolvable for L/U-PTAs.*

The full class of parametric timed automata The undecidability of the emptiness problems for L/U-PTAs proved above (Theorem 12) immediately implies undecidability for the larger class of PTAs. However, as in Remark 4, the full proof (given in [8]) of the result stated below uses less clocks and parameters than for L/U-PTAs (Theorem 12).

Theorem 13 (Undecidability of full (resp. weak) exp-ET-opacity Δ-p-emptiness problem [8, Theorem 7])**.** *The full (resp. weak) exp-ET-opacity Δ-p-emptiness problem is undecidable for general PTAs.*

Again, the synthesis problems are therefore immediately unsolvable as well.

Corollary 8 ([8, Corollary 3])**.** *The full (resp. weak) exp-ET-opacity Δ-p-synthesis problem is unsolvable for PTAs.*

6 Implementation and application to Java programs

A motivation for the works on ET-opacity (described in Sections 3 and 4) is the analysis of programs. More precisely, we are interested in deciding whether a program, e.g., written in Java, is ET-opaque, i.e., whether an attacker is incapable of deducing internal behavior by only looking at its execution time. A second motivation is the *configuration* of internal timing values from a program, e.g., changing some internal delays, or tuning some `Thread.sleep()` statements in the program, so that the program becomes ET-opaque—justifying notably the results in Section 4.

Semi-algorithm and implementation Despite the negative theoretical results (notably Theorem 1), we addressed in [9] the $\exists$-ET-opacity p-synthesis problem for the full class of PTAs. Our method may not terminate (due to the undecidability) but, if it does, its result is correct. Our workflow [9] can be summarized as follows.

1. We slightly modify the original PTA (by adding a Boolean flag b and a final synchronization action);
2. We perform *self-composition* (i.e., parallel composition with a copy of itself) of this modified PTA, a method commonly used in security analyses [32, 15];
3. We perform reachability-synthesis (i.e., the synthesis of parameter valuations for which a given location is reachable) on ℓ_f with contradictory values of b.

Reachability-synthesis is implemented in IMITATOR [6], a parametric timed model checker taking as inputs networks of (extensions of) parametric timed automata, and synthesizing parameter valuations for which a number of properties (including reachability) hold.

Analysis of Java programs In addition, we are interested in analyzing programs too. In order to apply our method to the analysis of programs, we need a systematic way of translating a program (e.g., a Java program) into a PTA. In general, precisely modeling the execution time of a program using models like TA is highly non-trivial due to complication of hardware pipelining, caching, OS scheduling, etc. The readers are referred to the rich literature in, e.g., [29, 20]. In [9], we instead make the following simplistic assumption on execution time of a program statement and focus on solving the parameter synthesis problem. We assume that the execution time of a program statement other than `Thread.sleep(n)` is within a range $[0, \varepsilon]$ where ε is a small integer constant (in milliseconds), whereas the execution time of statement `Thread.sleep(n)` is within a range $[n, n+\varepsilon]$. In fact, we choose to keep ε *parametric* to be as general as possible, and to not depend on particular architectures.

Our test subject is a set of benchmark programs from the DARPA Space/Time Analysis for Cybersecurity (STAC) program.[1] These programs are being released publicly to facilitate researchers to develop methods and tools for identifying STAC vulnerabilities in the programs. These programs are simple yet non-trivial, and were built on purpose to highlight vulnerabilities that can be easily missed by existing security analysis tools. We *manually* translated these programs to PTAs, following the method described above, and using a number of assumptions (such as collapsing loops with predefined duration).

In addition, we applied our method to a set of PTAs examples from the literature, notably from [27, 24, 16, 34].

Experiments reported in [9] show that we can decide whether these benchmarks (including the programs) are fully ET-opaque or $\exists$-ET-opaque. When adding timing parameters, we additionally answer the $\exists$-ET-opacity p-synthesis problem, i.e., we synthesize the parameter valuations v and the associated execution times D such that $v(\mathscr{P})$ is ET-opaque. Our method allows to exhibit cases when the system can never be made ET-opaque, including by tuning internal delays, or is always ET-opaque, or is ET-opaque only for some execution times and internal timing parameters.

To summarize, the following problems can be answered using our framework:

- $\exists$-ET-opacity decision problem
- full ET-opacity decision problem
- weak ET-opacity decision problem (not considered in our experiments in [9], but can be easily adapted)
- $\exists$-ET-opacity p-synthesis problem, but without guarantee of termination, due to the undecidability of Theorem 1.

However, our procedure cannot in its current form answer neither the full ET-opacity p-synthesis problem nor the weak ET-opacity p-synthesis problem. The expiring opacity problems in Section 5 were not addressed either.

7 Conclusion and perspectives

In this paper, we recalled (and proved a few original) results related to the ET-opacity in TAs. Our notion of ET-opacity consists in considering an attacker model that can only observe the execution time of the system, i.e., the time from the initial location to a final location. The secret consists in deciding whether a special private location was visited or not. In contrast to another notion of opacity with a more powerful attacker able to observe some actions together with their timestamps, which led to the undecidability of the decision problem for TAs [19], our notion of ET-opacity yields decidability results for TAs. Parameterizing the problems using timing parameters brings undecidability for PTAs, but the subclass of L/U-PTAs gives mildly positive results.

When in addition we consider that the secret has an expiration date, similarly to the concepts introduced in [4], we are able to not only *decide* problems for TAs, but also to *synthesize* valuations for the expiration date such that the TA is weakly exp-ET-opaque. However, problems extended with timing parameters all become undecidable.

Recall that we summarized in Tables 1 and 2 the decidability results recalled in this paper, with a bold emphasis on the original results of this paper.

[1]`https://github.com/Apogee-Research/STAC/`

We also reported here on an implementation using IMITATOR, which is able to answer non-parametric problems (∃-ET-opacity decision problem, full ET-opacity decision problem, weak ET-opacity decision problem), and also answering a parameter synthesis problem (∃-ET-opacity p-synthesis problem) without guarantee of termination for the latter problem.

Perspectives The main theoretical future work is the open problems in Table 2 (mainly the full exp-ET-opacity Δ-computation problem): it is unclear whether we can *compute* the exact set of expiration dates Δ for which a TA is fully $(\leq \Delta)$-ET-opaque.

In terms of synthesis, we have so far no procedure able (whenever it terminates) to answer the full ET-opacity p-synthesis problem or the weak ET-opacity p-synthesis problem. Synthesis procedures to answer expiring opacity problems (defined in Section 5) for PTAs remain to be designed too. These procedures cannot be both exact and guaranteed to terminate due to the aforementioned undecidability results.

Exact analysis of opacity for programs, including a more precise modeling of the cache, is also on our agenda, following works such as [20, 21].

A different direction is that of *control*: can we turn a non-opaque system into an opaque system, by restraining its possible behaviors? A first step with our notion of ET-opacity was presented in [7], with only an *untimed* controller. In addition, in [24], Gardey *et al.* propose several definitions of non-interference, related to various notions of simulation: they consider not only the *verification* problem ("is the system non-interferent?") but also the (timed) *control* problem ("synthesize a controller that will restrict the system in order to enforce non-interference"). Extending our current line works on ET-opacity to *timed* controllers remains to be done.

Acknowledgments We are grateful to Clemens Dubslaff and Maurice ter Beek for the opportunity to give an invited talk at TiCSA 2023, and for useful suggestions on this manuscript.

References

[1] Rajeev Alur & David L. Dill (1994): *A theory of timed automata*. Theoretical Computer Science 126(2), pp. 183–235, doi:10.1016/0304-3975(94)90010-8.

[2] Rajeev Alur, Limor Fix & Thomas A. Henzinger (1999): *Event-Clock Automata: A Determinizable Class of Timed Automata*. Theoretical Computer Science 211(1-2), pp. 253–273, doi:10.1016/S0304-3975(97)00173-4.

[3] Rajeev Alur, Thomas A. Henzinger & Moshe Y. Vardi (1993): *Parametric real-time reasoning*. In S. Rao Kosaraju, David S. Johnson & Alok Aggarwal, editors: STOC, ACM, New York, NY, USA, pp. 592–601, doi:10.1145/167088.167242.

[4] Ikhlass Ammar, Yamen El Touati, Moez Yeddes & John Mullins (2021): *Bounded opacity for timed systems*. Journal of Information Security and Applications 61, pp. 1–13, doi:10.1016/j.jisa.2021.102926.

[5] Étienne André (2019): *What's decidable about parametric timed automata?* International Journal on Software Tools for Technology Transfer 21(2), pp. 203–219, doi:10.1007/s10009-017-0467-0.

[6] Étienne André (2021): *IMITATOR 3: Synthesis of timing parameters beyond decidability*. In Rustan Leino & Alexandra Silva, editors: CAV, Lecture Notes in Computer Science 12759, Springer, pp. 1–14, doi:10.1007/978-3-030-81685-8_26.

[7] Étienne André, Shapagat Bolat, Engel Lefaucheux & Dylan Marinho (2022): *strategFTO: Untimed control for timed opacity*. In Cyrille Artho & Peter Ölveczky, editors: *FTSCS*, ACM, pp. 27–33, doi:10.1145/3563822.3568013.

[8] Étienne André, Engel Lefaucheux & Dylan Marinho (2023): *Expiring opacity problems in parametric timed automata*. In Yamine Ait-Ameur & Ferhat Khendek, editors: *ICECCS*. To appear.

[9] Étienne André, Didier Lime, Dylan Marinho & Jun Sun (2022): *Guaranteeing timed opacity using parametric timed model checking*. *ACM Transactions on Software Engineering and Methodology* 31(4), pp. 1–36, doi:10.1145/3502851.

[10] Étienne André, Didier Lime & Nicolas Markey (2020): *Language Preservation Problems in Parametric Timed Automata*. *Logical Methods in Computer Science* 16(1), doi:10.23638/LMCS-16(1:5)2020. Available at `https://lmcs.episciences.org/6042`.

[11] Étienne André, Didier Lime & Mathias Ramparison (2018): *TCTL model checking lower/upper-bound parametric timed automata without invariants*. In David N. Jansen & Pavithra Prabhakar, editors: *FORMATS*, *Lecture Notes in Computer Science* 11022, Springer, pp. 1–17, doi:10.1007/978-3-030-00151-3_3.

[12] Étienne André, Didier Lime & Olivier H. Roux (2022): *Reachability and liveness in parametric timed automata*. *Logical Methods in Computer Science* 18(1), pp. 31:1–31:41, doi:10.46298/lmcs-18(1:31)2022. Available at `https://lmcs.episciences.org/9070/pdf`.

[13] Johan Arcile & Étienne André (2023): *Timed automata as a formalism for expressing security: A survey on theory and practice*. *ACM Computing Surveys* 55(6), pp. 1–36, doi:10.1145/3534967.

[14] Roberto Bagnara, Patricia M. Hill & Enea Zaffanella (2008): *The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems*. *Science of Computer Programming* 72(1–2), pp. 3–21, doi:10.1016/j.scico.2007.08.001.

[15] Gilles Barthe, Pedro R. D'Argenio & Tamara Rezk (2011): *Secure information flow by self-composition*. *Mathematical Structures in Computer Science* 21(6), pp. 1207–1252, doi:10.1017/S0960129511000193.

[16] Gilles Benattar, Franck Cassez, Didier Lime & Olivier H. Roux (2015): *Control and synthesis of non-interferent timed systems*. *International Journal of Control* 88(2), pp. 217–236, doi:10.1080/00207179.2014.944356.

[17] Laura Bozzelli & Salvatore La Torre (2009): *Decision problems for lower/upper bound parametric timed automata*. *Formal Methods in System Design* 35(2), pp. 121–151, doi:10.1007/s10703-009-0074-0.

[18] Véronique Bruyère, Emmanuel Dall'Olio & Jean-Francois Raskin (2008): *Durations and parametric model-checking in timed automata*. *ACM Transactions on Computational Logic* 9(2), pp. 12:1–12:23, doi:10.1145/1342991.1342996.

[19] Franck Cassez (2009): *The Dark Side of Timed Opacity*. In Jong Hyuk Park, Hsiao-Hwa Chen, Mohammed Atiquzzaman, Changhoon Lee, Tai-Hoon Kim & Sang-Soo Yeo, editors: *ISA*, *Lecture Notes in Computer Science* 5576, Springer, pp. 21–30, doi:10.1007/978-3-642-02617-1_3.

[20] Franck Cassez & Jean-Luc Béchennec (2013): *Timing Analysis of Binary Programs with UPPAAL*. In Josep Carmona, Mihai T. Lazarescu & Marta Pietkiewicz-Koutny, editors: *ACSD*, IEEE Computer Society, pp. 41–50, doi:10.1109/ACSD.2013.7.

[21] Duc-Hiep Chu, Joxan Jaffar & Rasool Maghareh (2016): *Precise Cache Timing Analysis via Symbolic Execution*. In: *RTAS*, IEEE Computer Society, pp. 293–304, doi:10.1109/RTAS.2016.7461358.

[22] Shuwen Deng, Wenjie Xiong & Jakub Szefer (2018): *Cache timing side-channel vulnerability checking with computation tree logic*. In Jakub Szefer, Weidong Shi & Ruby B. Lee, editors: *ISCA*, ACM, pp. 2:1–2:8, doi:10.1145/3214292.3214294.

[23] Goran Doychev, Boris Köpf, Laurent Mauborgne & Jan Reineke (2015): *CacheAudit: A Tool for the Static Analysis of Cache Side Channels*. *ACM Transactions on Information and System Security* 18(1), pp. 4:1–4:32, doi:10.1145/2756550.

[24] Guillaume Gardey, John Mullins & Olivier H. Roux (2007): *Non-Interference Control Synthesis for Security Timed Automata*. *Electronic Notes in Theoretical Computer Science* 180(1), pp. 35–53, doi:10.1016/j.entcs.2005.05.046.

[25] Shengjian Guo, Meng Wu & Chao Wang (2018): *Adversarial symbolic execution for detecting concurrency-related cache timing leaks*. In Gary T. Leavens, Alessandro Garcia & Corina S. Pasareanu, editors: *ESEC/SIGSOFT FSE*, ACM, pp. 377–388, doi:10.1145/3236024.3236028.

[26] Thomas A. Henzinger, Zohar Manna & Amir Pnueli (1992): *Timed Transition Systems*. In J. W. de Bakker, Cornelis Huizing, Willem P. de Roever & Grzegorz Rozenberg, editors: *REX*, *Lecture Notes in Computer Science* 600, Springer, pp. 226–251, doi:10.1007/BFb0031995.

[27] Thomas Hune, Judi Romijn, Mariëlle Stoelinga & Frits W. Vaandrager (2002): *Linear parametric model checking of timed automata*. *Journal of Logic and Algebraic Programming* 52-53, pp. 183–220, doi:10.1016/S1567-8326(02)00037-1.

[28] Aleksandra Jovanović, Didier Lime & Olivier H. Roux (2015): *Integer Parameter Synthesis for Real-Time Systems*. *IEEE Transactions on Software Engineering* 41(5), pp. 445–461, doi:10.1109/TSE.2014.2357445.

[29] Mingsong Lv, Wang Yi, Nan Guan & Ge Yu (2010): *Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software*. In: *RTSS*, IEEE Computer Society, pp. 339–349, doi:10.1109/RTSS.2010.30.

[30] Joseph S. Miller (2000): *Decidability and Complexity Results for Timed Automata and Semi-linear Hybrid Automata*. In Nancy A. Lynch & Bruce H. Krogh, editors: *HSCC*, *Lecture Notes in Computer Science* 1790, Springer, pp. 296–309, doi:10.1007/3-540-46430-1_26.

[31] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria & Tevfik Bultan (2017): *Synthesis of Adaptive Side-Channel Attacks*. In: *CSF*, IEEE Computer Society, pp. 328–342, doi:10.1109/CSF.2017.8.

[32] Tachio Terauchi & Alexander Aiken (2005): *Secure Information Flow as a Safety Problem*. In Chris Hankin & Igor Siveroni, editors: *Proceedings of the 12th International Symposium on Static Analysis (SAS 2005)*, *Lecture Notes in Computer Science* 3672, Springer, pp. 352–367, doi:10.1007/11547662_24.

[33] Saeid Tizpaz-Niari, Pavol Cerný & Ashutosh Trivedi (2019): *Quantitative Mitigation of Timing Side Channels*. In Işil Dillig & Serdar Tasiran, editors: *CAV, Part I*, *Lecture Notes in Computer Science* 11561, Springer, pp. 140–160, doi:10.1007/978-3-030-25540-4_8.

[34] Panagiotis Vasilikos, Flemming Nielson & Hanne Riis Nielson (2018): *Secure Information Release in Timed Automata*. In Lujo Bauer & Ralf Küsters, editors: *POST*, *Lecture Notes in Computer Science* 10804, Springer, pp. 28–52, doi:10.1007/978-3-319-89722-6_2.

[35] Panagiotis Vasilikos, Hanne Riis Nielson, Flemming Nielson & Boris Köpf (2019): *Timing Leaks and Coarse-Grained Clocks*. In: *CSF*, IEEE, pp. 32–47, doi:10.1109/CSF.2019.00010.

[36] Lingtai Wang & Naijun Zhan (2018): *Decidability of the Initial-State Opacity of Real-Time Automata*. In Cliff B. Jones, Ji Wang & Naijun Zhan, editors: *Symposium on Real-Time and Hybrid Systems - Essays Dedicated to Professor Chaochen Zhou on the Occasion of His 80th Birthday*, *Lecture Notes in Computer Science* 11180, Springer, pp. 44–60, doi:10.1007/978-3-030-01461-2_3.

[37] Lingtai Wang, Naijun Zhan & Jie An (2018): *The Opacity of Real-Time Automata*. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37(11), pp. 2845–2856, doi:10.1109/TCAD.2018.2857363.

[38] Meng Wu, Shengjian Guo, Patrick Schaumont & Chao Wang (2018): *Eliminating timing side-channel leaks using program repair*. In Frank Tip & Eric Bodden, editors: *ISSTA*, ACM, pp. 15–26, doi:10.1145/3213846.3213851.

Spreadsheet-based Configuration of Families of Real-Time Specifications

José Proença
CISTER and University of Porto, Portugal
jose.proenca@fc.up.pt

David Pereira Giann Spilere Nandi
CISTER, Polytechnic Institute of Porto, Portugal
{drp,giann}@isep.ipp.pt

Sina Borrami Jonas Melchert
Alstom
{sina.borrami,jonas.melchert}@alstomgroup.com

1 Introduction

Model checking real-time systems is complex. This particular work was motivated by and developed in collaboration with an industrial use-case provider: the Alstom railway company, in the context of the VALU3S European project. In this use-case we formally analyse a motor controller used in signalling systems: a safety-critical embedded system that reacts to instructions to turn a motor left or right. Given the criticality of this system and the need to comply to railway standards [11, 12, 13], the motor controller includes redundancy techniques, and its certification requires formal evidences that given time-bounds are met.

The implementation of this motor controller has been developed hand-in-hand with the formal specification of a real-time model in Uppaal [9], with a mutual influence between the two. Full details of this use-case can be found in our previous work [16]. The level of detail and the amount of non-determinism in early models quickly led to state-space explosions when analysing properties such as deadlock freedom. To cope with the state-space explosion problem, different details could be abstracted away. This led us to two core challenges: (i) how to efficiently involve both experts in model-checking and experts in the application domain; and (ii) how to balance trade-offs in the formal specifications between including *enough details* to be faithful to the implementation and *not too many details* to avoid model-checking more complex requirements.

Our approach involves the creation of many variations of the real-time specification, and using MS Excel spreadsheets to help keeping the developers engaged and not interacting directly with the model-checker. The Uppaal specifications are annotated, and a set of companion spreadsheets controls variability, i.e., for each variation it configures both how the annotated parts of the Uppaal specification can be modified and which requirements should be used.

Contributions. This paper presents extensions that provide a better support for variability, introducing the concept of a feature model [18] within the spreadsheets to validate configurations, and introducing integer attributes to these feature models. We provide a companion open-source tool—`Uppex`—that reads MS Excel spreadsheets and Uppaal models and automatises the feature analysis and the model-checking processes. The results are validated within the railway use-case, provided by Alstom, already described in detail in our previous work [16]. We further use a simpler example that the reader can use to experiment with `Uppex`.

Related work Model-checking complex systems is difficult and often infeasible due to space explosion. A possible approach to verify properties over networks of automata with a state-space that is too large to traverse is to use statistical model checking (SMC) [15]. Uppaal Stratego supports SMC [9], and has shown promising results in the railway domain over a moving block signalling system [3]. Using SMC, properties are quantified over the probability of occurring, and model-checking involves performing many runs of the system until the confidence reflects the probability of the property. Uppex provides an alternative to model-check complex systems, without losing the strength of symbolic model-checking, by facilitating the process of producing many simplifications, each abstracting over different aspects. This family of simpler models is automatically model-checked by successive instantiations and invocations to Uppaal. Although we use the Uppaal model checker, this tool and our methodology can be easily adapted to other model-checkers such as IMITATOR [1] or mCRL2 [6].

The idea of verifying a family of systems efficiently has been investigated and well received in the software product line community [8, 6]. The goal of these approaches is to be able to verify a set of properties in all members of a family of systems. This is often realised by modelling the variability aspect together with the behavioural aspect, avoiding the generation of one model for each member. On the contrary, our approach produces one instance of the model for each member. This creates less dependencies to the choice of the concrete model-checker and allows customising which properties are verified at each instance, at the cost of performance and number of configurations supported. Furthermore, Uppex attempts to provide an easy interface between modellers and developers, giving the power to developers to fine-tune parameters and configurations without being exposed to the model-checker.

Uppex uses a Domain Specific Language to represent feature models, for which many textual and modelling languages exist [5]. A feature model is here represented as a spreadsheet table, getting inspiration mainly from the UVL language [19], but exploiting the tabular representation to capture the tree structure of feature diagrams [18].

Several approaches exist to realise variability, i.e., to generate software artefacts from a selection of features [10]. Popular ones include annotative and compositional approaches [2]. Annotative approaches mark code blocks that should be removed when some feature is absent at compile time, e.g. using the C-preprocessor to hide blocks of code using `#ifdef` directives. Compositional approaches, such as feature-oriented programming [4], aspect-oriented programming [14], and delta-oriented programming [17], provide mechanisms to inject blocks of code based on the selected features. Uppex uses annotated blocks in a compositional way, i.e., they act as *hooks* marking consecutive lines of the specification file that can be modified when producing variations. This is aligned with the aspect-oriented approach, which uses patterns to discover blocks to be adapted (instead of explicit hooks), and with the delta-oriented approach, which uses the names of structural elements (such as classes, objects, and methods) as the blocks to be adapted. Our approach is more primitive, in the sense that it is not aware of the structure of the documents being adapted. This makes it more independent of the target language and analyser being used in the back-end, at the cost of understanding and reusing the content of the blocks being replaced. For example, Uppex cannot keep an existing annotated block and add a new line, but can only replace the full block with a new one.

Organization of the paper. Section 2 provides more details over our motivating railway scenario prior to our extensions. Section 3 describes how to add variability to Uppaal models with Uppex, using features an feature models, using a simpler example. Section 4 summarises some lessons learned when using Uppex, and Section 5 concludes this paper and suggests lines of future work.

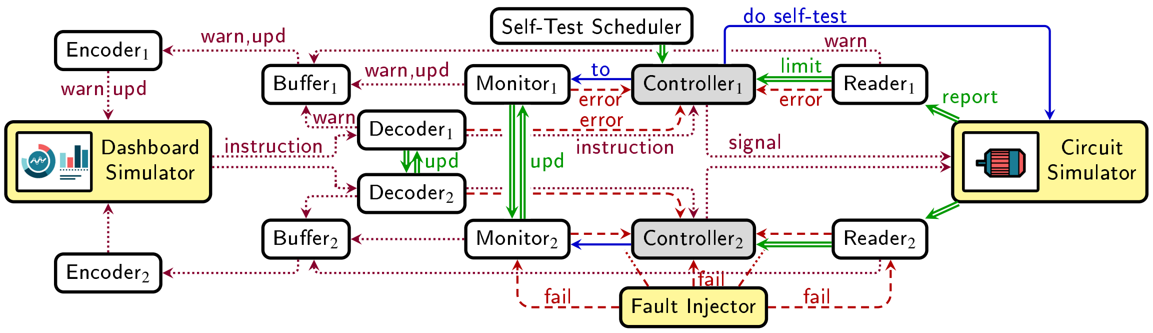


Figure 1: Architecture of the concurrent components being modelled.

2 Motivation: model-checking a motor controller

The system under study is a motor controller; its detailed component architecture is depicted in Fig. 1. Overall, the controller receives *instructions* from a dashboard (to turn left, to turn right, or a heartbeat), and sends *signals* to a circuit that triggers the corresponding rotation of an engine. The circuit sends periodic *reports* to the controller, either informing that the maximum rotation was reached or that a problem was found. Finally the controller notifies the dashboard whenever an important update or warning exists.

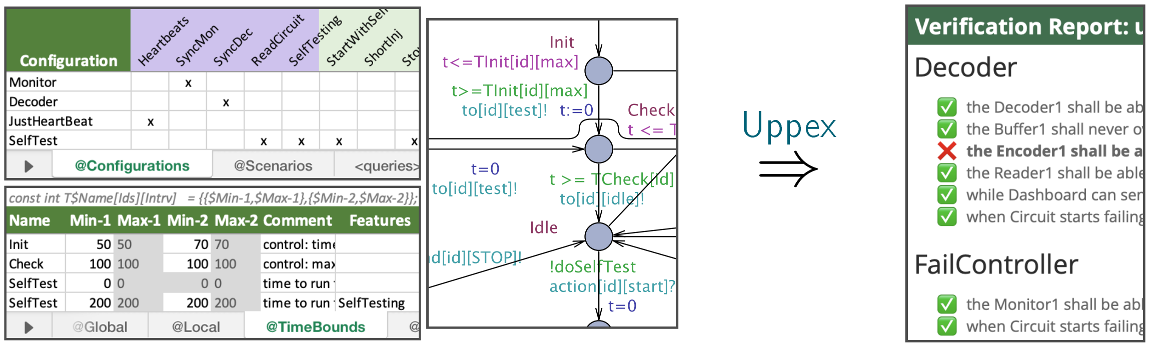
The architecture in Fig. 1 includes other details, explained below.

- The system has redundancy: most components are replicated (e.g., Controller$_1$ and Controller$_2$, and their consistency is verified by monitors and decoders.
- The environment is modelled by 3 components: the Dashboard Simulator, the Circuit Simulator, and the Fault Injector; different scenarios can be considered, to analyse the behaviour under well- and ill-behaved environments.
- The components interact in different ways: using synchronisation barriers ($\longrightarrow$), non-blocking synchronous sends that lose data when the reader is not ready ($\dashrightarrow$) or that are guaranteed by the receiver to be received ($\dashrightarrow$), and asynchronous interaction via a shared variable that is written by the sender and read by the receiver ($\Longrightarrow$).

The core behaviour is described by both Controller components, who are responsible to detect errors and enter a fallback state in such cases, e.g., when the engines take too long or are too fast to reach the end of a rotation.

Our formal model of this system in Uppaal encodes each component as a state machine, more specifically a real-time automaton [9]. When model-checking this model many requirements cannot be verified precisely due to a space explosion. This is because, in many time-points, a very large number of interleavings were possible. E.g., often 8 different components could perform some interaction in any possible order. Our solution consists in creating many *variants* of the real-time model, simplifying different aspects of this model, and selecting different requirements to different variants. These variants include, among others:

- different environments (dashboards, circuits, and fault injectors);
- discarded heartbeat signals, i.e., periodic messages sent from the dashboard to confirm that the motor is available;
- discarded consistency checks between replicated counter-parts;



Configuration	Heartbeats	SyncMon	SyncDec	ReadCircuit	SelfTesting	StartWithSel	ShortInj	Sto
Monitor		x						
Decoder			x					
JustHeartBeat	x							
SelfTest				x	x	x		x

▶ @Configurations @Scenarios <queries>

const int T$Name[Ids][Intrv] = {{$Min-1,$Max-1},{$Min-2,$Max-2}};

Name	Min-1	Max-1	Min-2	Max-2	Comment	Features
Init	50	50	70	70	control: tim	
Check	100	100	100	100	control: ma	
SelfTest	0	0	0	0	time to run	
SelfTest	200	200	200	200	time to run	SelfTesting

▶ @Global @Local @TimeBounds @

Verification Report: u

Decoder

- ✅ the Decoder1 shall be ab
- ✅ the Buffer1 shall never o
- ❌ **the Encoder1 shall be a**
- ✅ the Reader1 shall be able
- ✅ while Dashboard can ser
- ✅ when Circuit starts failing

FailController

- ✅ the Monitor1 shall be abl
- ✅ when Circuit starts failing

Figure 2: Uppex workflow: updating and verifying models based on configuration tables

- discarded reading from the circuit; and
- discarded part of the controller behaviour (initial tests).

In total, we collected ±35 aspects that could be toggled, called *features*, and manually selected 14 combinations of these features, called *configurations*. The choice of this combinations was driven by the requirements, i.e., adapted until each requirement could be verified in a rich-enough set of variants. Note that some of these features were describing requirements that must be verified, e.g., if deadlock freedom should be verified. Also note that these features are not meant to be optional features of the implementation, e.g., we always expect the final system to use heartbeats; however, abstracting it away in some variations allows the verification of properties that do not rely on heartbeats.

Statistical Model Checking (SMC), also supported by Uppaal and applied in a similar context [3], is an alternative approach that we avoid. Using SMC one can verify properties with a given level of certainty, based on many runs of the model. However, it does not provide the same level of certainty of traditional symbolic model-checking.

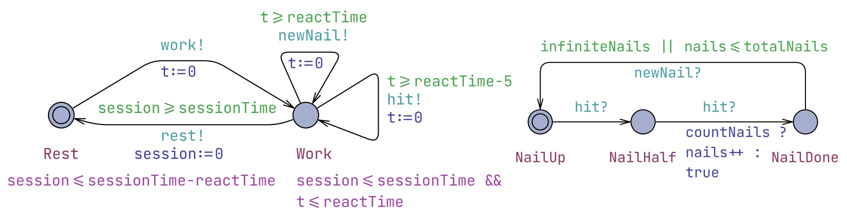
Automatisation with spreadsheets and Uppex

We propose to automatise the verification of these variants, initially reported in RSSRail 2022 [16], using (i) *spreadsheets* to represent both core parameters and requirements of the system under study, and (ii) a prototype tool Uppex[1] to automatise the creation and verification of variations of the formal specification, whereas each variation can have a different set of requirements. Formal models are annotated, specified, and verified using the Uppaal model checker [9]. Uppaal targets real-time systems, using special variables called clocks that capture the passage of time, and using these clocks to guide the behaviour (with some syntactic restrictions that make the model-checking problem feasible).

Uppex is an open-source command-line tool developed in Scala that reads both a set of spreadsheets with configurations in MS Excel and an annotated Real-Time specification in Uppaal. Other back-ends are future work, e.g. IMITATOR [1]. A typical workflow is depicted in Fig. 2: given a set of configuring spreadsheets and an annotated Uppaal specification (left), Uppex produces an `html` report (right) listing properties that passed, failed, or timed-out for each configuration.

More specifically, Uppex interprets (1) *special sheets from a MS Excel file* and (2) an *annotated Uppaal file* (XML format), briefly described below.

[1]https://cister-labs.github.io/uppex/



```
int nails = 0; // number of completed nc

// @Limits
const int sessionTime = 100; // Total ti
const int reactTime = 20; // Maximum tin
const bool countNails = true; // If the
const int totalNails = 0; // Total numbe
const bool infiniteNails = true; // If t
```

Figure 3: Annotated Uppaal specification of a `worker` (left) and a `hammer` (middle); this specification is an XML file with code snippets (c.f. right side) with c-like code that is used by the automata

- Two types of *annotated blocks* are recognised by `Uppex` in the Uppaal file: (1) a sequence of consecutive lines starting with "`// @BlockName`" until an empty line, such as the "`// @Limit`" block on the right of Fig. 3, and (2) an XML element "`<BlockName>...</BlockName>`", covering the text between the tags. Both these annotated blocks have an identifier (the `BlockName`) and a consecutive sequence of lines.
- The sheet `@Configurations` (top-left of Fig. 2) lists valid combinations of *features*, describing *configuration names* in the first column and *feature names* in the first row.
- Any other sheet starting with `@`, such as `@Timebounds` (bottom-left of Fig. 2) describe what code will be injected in the Uppaal specification, in this case in an annotated block named `Timebounds`. The column named `Features` is used to filter rows based on the selected configuration – in this case the last two rows have the same identifier (`SelfTest`), and when the `SelfTesting` feature is active the last row will override the previous one. We call these *@-annotations*.
- Any sheet with a name surrounded by `<·>` (e.g. `<queries>`), is similar to an `@`-sheet, but targetting annotated blocks given by XML elements, as explained above. We call these *xml-annotations*.

We will describe each of these tables and annotations in more detail below, guided by a simpler example, and extend this approach to further exploit the analyses of features.

3 Feature modelling in `Uppex`

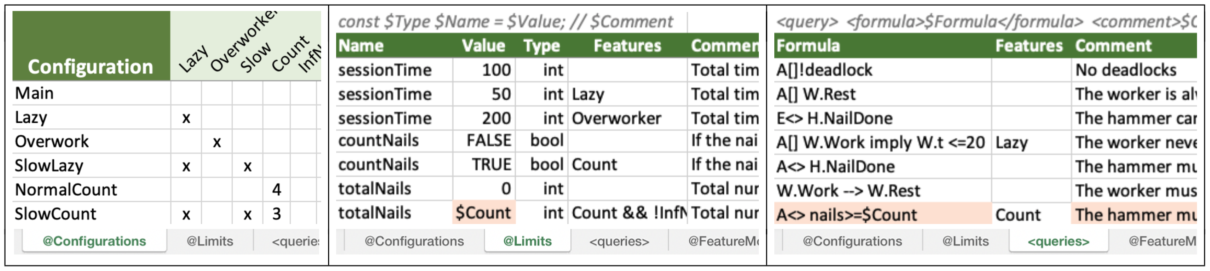
In this work we extend `Uppex` to further exploit the feature analysis, introducing *data attributes*, *feature conditions*, and a *feature model*. These are explained below using a simpler but complete example of a `hammer` automaton interacting with a `worker` automata while hitting nails. This example can be found together with the tool at `https://github.com/cister-labs/uppex/blob/v0.1.3/examples`.

3.1 Annotating Uppaal specifications

When developing a family of models with `Uppex`, the starting point is a parameterised model. In Fig. 3 we present a simple example with 2 timed-automata, where a `worker` is either `Rest`ing or `Work`ing. While working, it uses a `hammer` to either `hit` a nail or to place a `newNail`. The code on the right side is used by the Uppaal specification; e.g., `sessionTime` represents the combined time to rest and work by the `worker`, set to 100. The other variables, from top to bottom respectively, capture the maximum time to hit a nail or to add a new one, if the nails should be counted, the number of nails, and if no limit of nails should be considered. The details of the semantics of timed-automata are out of the scope of this paper; intuitively each transition can have a `guard` representing when the transition is active, an `action` that will act as a synchronisation barrier with a counterpart action, and an `update` that updates variables after a transition. Some special variables represent time and are called *clocks*; in our example `t` and `session`.

3.2 Configuring variants

A *configuration* is a variation of the Uppaal specification by replacing an annotated block by a new block with the same name. In our example, the code on the right of Fig. 3 has a `@Limits` block with 5 lines. Using a companion MS Excel spreadsheet, we can specify configurations that describe how these annotated blocks can be replaced.



Configuration	Lazy	Overworke	Slow	Count	InfN
Main					
Lazy	x				
Overwork		x			
SlowLazy	x		x		
NormalCount				4	
SlowCount	x		x	3	

const $Type $Name = $Value; // $Comment

Name	Value	Type	Features	Commen
sessionTime	100	int		Total tim
sessionTime	50	int	Lazy	Total tim
sessionTime	200	int	Overworker	Total tim
countNails	FALSE	bool		If the nai
countNails	TRUE	bool	Count	If the nai
totalNails	0	int		Total nur
totalNails	$Count	int	Count && !InfN	Total nur

<query> <formula>$Formula</formula> <comment>$C

Formula	Features	Comment
A[]!deadlock		No deadlocks
A[] W.Rest		The worker is al
E<> H.NailDone		The hammer car
A[] W.Work imply W.t <=20	Lazy	The worker neve
A<> H.NailDone		The hammer mu
W.Work --> W.Rest		The worker mus
A<> nails>=$Count	Count	The hammer mu

Figure 4: Defining configurations with spreadsheets: selection of features in `@Configuration` (left), defining the `@Limits` annotation (middle), and defining the `<queries>` annotation (right)

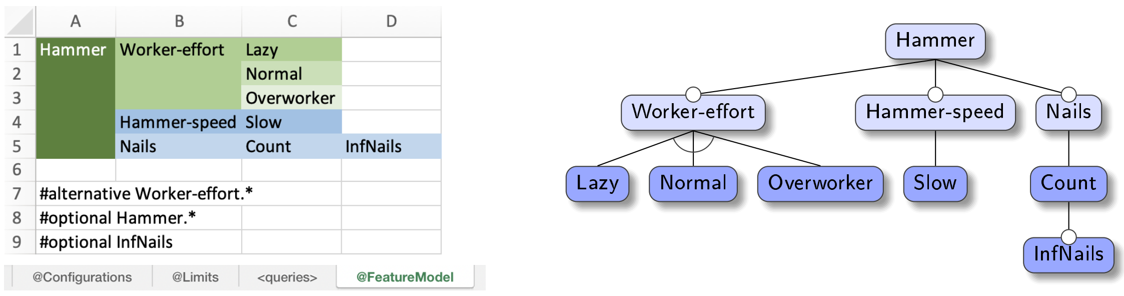
The middle of Fig. 4 presents the `@Limits` sheet in our hammer example, containing a table of values that is used to produce the associated `@Limits` annotation block.

This table is called an *@-annotation*. In the new block each line is formatted according to the top row "*`const $Type $Name = $Value; // $Comment`*". Blocks can also refer to XML tags, to replace blocks delimited by a given tag; e.g. the sheet on the right of Fig. 4 is an *xml-annotation* that specifies a list of requirements using Uppaal's logic that will replace the content of the `<queries>` XML element.

A configuration is a set of features, defined in the `@Configuration` table (left of Fig. 4). For example, the configuration named SlowLazy includes the features Lazy and Slow. Features can also have an associated value, e.g., Count is assigned to "4" in configuration NormalCount and to "3" in SlowCount.

The annotation tables (c.f. middle and right of Fig. 4) can have a special column named `Features` with boolean expressions over feature names. This is used to filter rows: given a configuration, only rows with an expression that holds for the corresponding set of feature is considered. Empty expressions are trivially true. Furthermore, the left-most column acts as an identifier: if more than a row with the same identifier is selected, the last one with a valid feature expression is used. We chose to use an *overriding* interpretation, instead of forcing these feature expressions to be disjoint for entries with the same identifier, because we found these specifications to be simpler to write and more compact. In this example selecting the Overworker feature and not Lazy will discard the 2nd row for sessionTime, and the 3rd will override the 1st. I.e., the variable sessionTime will be set to 200. The value of a feature can be used in the other cells of a row; e.g., the value of totalNails will be set to 4 when choosing the configuration NormalCount.

This work extends our previous approach [16] by (i) associating values to features and (ii) using of expressions over features instead of individual features in the Features column.



	A	B	C	D
1	Hammer	Worker-effort	Lazy	
2			Normal	
3			Overworker	
4		Hammer-speed	Slow	
5		Nails	Count	InfNails
6				
7	#alternative Worker-effort.*			
8	#optional Hammer.*			
9	#optional InfNails			

@Configurations @Limits <queries> @FeatureModel

Figure 5: Example of a feature diagram: its tabular form (left) and its usual representation (right)

3.3 Validating features

Not all combination of features in the `@Configuration` table should be considered. For example, the worker should not be both lazy and overworker. Such constraints are compiled in another special table called `@FeatureModel`, using a tabular form of feature diagrams [18]. These constraints describe valid combinations of features but are not related to the `Feature` columns in the annotation tables. We borrowed some constructs from the textual UVL language for feature models [19], and synthesise UVL diagrams in Uppex. An example of a feature diagram can be found in Fig. 5: on the left our tabular representation, and on the right its more traditional visual representation. The table is interpreted as follows.

- Non-empty rows whose 1st column does not start with `#` describe the **tree structure**: the parents on the left and the children on the right. For example, cell `C4` (`Slow`) is the child of `B4` (`Hammer-speed`), which in turn is the child of `A4`; the latter cell is empty, meaning that it inherits the previous value in column `A`, i.e. `A1` (`Hammer`).

- Rows whose 1st column starts with `#` describe a **constraint**:

  – `#mandatory <siblings>` – given a set of features with a common parent (`siblings`), it states that these are mandatory whenever the parent is selected;

  – `#optional <siblings>` – states that a set of siblings are optional, even if the parent is selected

  – `#alternative <siblings>` – states that a set of siblings are exclusive and at most one should be included whenever the parent is selected;

  – `#or <siblings>` – states that at least one out of a set of siblings should be included whenever the parent is selected;

  – `#constraint <feature-constraint>` – is a boolean formula over features, following the same syntax as in the `Features` column (c.f. Section 3.2), that must hold.

Only the `#alternate` and the `#optional` constraints are illustrated in Fig. 5, and by default all features are mandatory. Combining the tree structure and the constraints yields a feature diagram, such as the one on the right of Fig. 5. Currently Uppex supports feature-constraints over features but not over feature attributes, which is left as future work. The tree structure also imposes a strong need to include parent features whenever a child is selected – Uppex exploits this by automatically expanding the selection of features to all the parents of the selected ones.

3.4 Workflow using `Uppex` and Uppaal

So far we described how to specify the input models: (i) the annotated Uppaal specification, (ii) the tables with possible parameters and requirements, (iii) the table with configurations of features, and (iv) the table with the feature model. This subsection describes our proposed *methodology*, i.e., the suggested workflow with `Uppex` and Uppaal during the development of a model.

`Uppex`'s tool is a standalone JAR file `uppex.jar`, open-source and available at `https://github.com/cister-labs/uppex/releases`, that can be executed as a command line tool using `java -jar uppex.jar [options] <mytables.xlsx>`. We expect a typical development of a Uppaal+`Uppex` project to proceed as follows.

1. **Model:** Produce a base Uppaal model `project.xml`, i.e., a network of timed automata that can be simulated in Uppaal.
   Edit: Automata in Uppaal

2. **Parameterise:** Identify a set of parameters that can be useful to expose to domain experts and create the associated @-annotations in the companion Excel file `project.xlsx`; update the Uppaal model by running `Uppex` with no arguments, e.g. `java -jar uppex.jar project.xlsx`.
   Edit: Automata in Uppaal & @-annotations in Excel

3. **Verify behaviour:** Identify a set of requirements, specify them using Uppaal's CTL, and place these in the `<queries>` spreadsheet (c.f. right of Fig. 4); update the Uppaal model as before, or verify all properties using `Uppex` using the command `java -jar uppex.jar --run project.xlsx`.
   Edit: `<queries>`-annotation in Excel

4. **Instantiate:** Identify variability points and features, populating the annotation tables with a column `Features` (c.f. right of Fig. 4); create the `@Configurations` table to list products, i.e., desired combinations of features (c.f. left of Fig. 4); transform the working Uppaal file to match any given configuration (or product) `prod` by running `java -jar uppex.jar -p prod project.xlsx`; the verification in step (3) can also receive the `-p prod` option, or simply `--runAll` to verify all available products.
   Edit: `@Configurations` & annotations in Excel

5. **Verify instances:** Identify restrictions over what features can be combined, and specify these in the special `@FeatureModel` table (c.f. Fig. 5); `Uppex` will always validate all features when verifying or applying a product, but can also be used exclusively for validation by running `java -jar --validate project.xlsx`.
   Edit: `@FeatureModel` in Excel

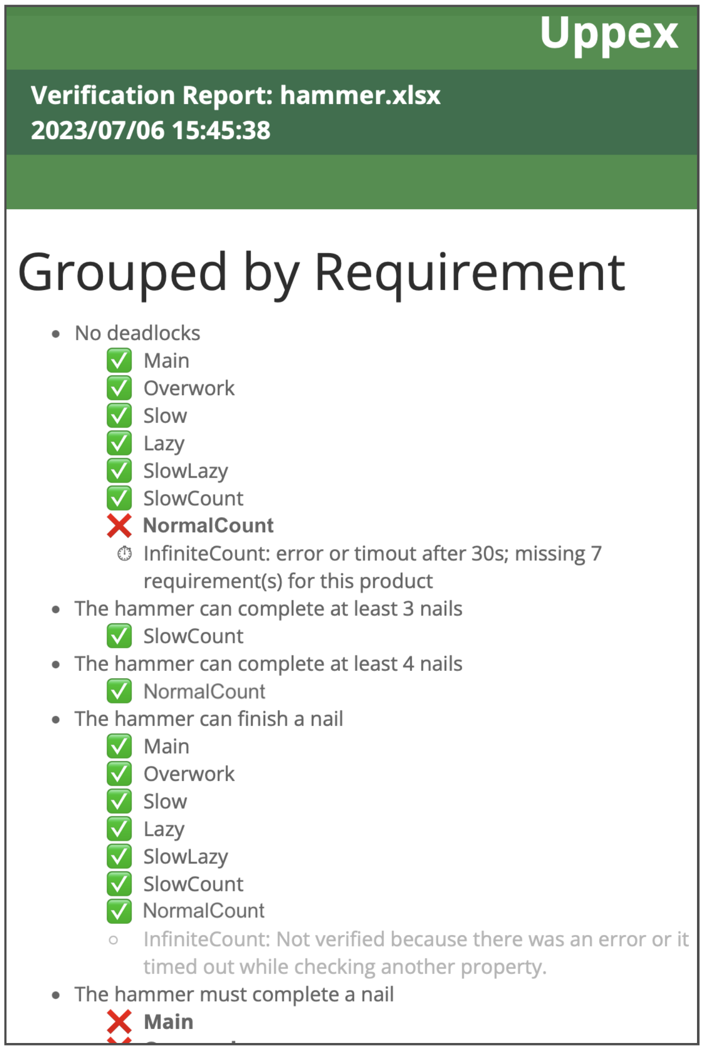
At each of the steps above it is often needed to revisit the previous steps. E.g., after the verification step (3) we expect to be needed to revisit the model in steps (1) and (2), to adapt it based on the verification results.

When a product is applied, a backup of the original version is stored in a folder `backups`, to prevent losing parameters by mistake. This resembles a naïve implementation of a version-control system, where applying a product modifies the working document, while keeping the history of previous versions.

Verifying properties with Uppaal requires the `verifyta` tool to be available at the command line, called by `Uppex` using system calls.[2] After verifying all properties of all products with `java -jar uppex.jar --runnAll project.xlsx`, the tool presents a summary of annotations and configurations found, the feature model in plain text using UVL [19], potential errors when validating products, and the

[2]Uppaal is a commercial tool, but freely available for academic partners.

```
>>> java -jar uppex.jar \
    --runAll hammer.xlsx
features
  Hammer
    optional
      Worker-effort
        alternative
          Lazy
          Normal
          Overworker
      Hammer-speed
        mandatory
          Slow
      Nails
        mandatory
          Count
            optional
              InfNails
constraints
  !(Lazy && Overworker)
---
 - Products: InfiniteCount,
     NormalCount, SlowCount,
     SlowLazy, Lazy, Slow, Overwork,
     Main
> Reading Uppaal file 'hammer.xml'
---Verifying 'InfiniteCount'---
  | Error or time-out after 30s.
      Missing 7 properties. Failed on:
  | "No_deadlocks"
---Verifying 'NormalCount'---
[FAIL] No deadlocks
[FAIL] The worker is always resting
[OK] The hammer can finish a nail
[FAIL] The hammer must complete a nail
...
```


Uppex

Verification Report: hammer.xlsx
2023/07/06 15:45:38

Grouped by Requirement

- No deadlocks
  - ✓ Main
  - ✓ Overwork
  - ✓ Slow
  - ✓ Lazy
  - ✓ SlowLazy
  - ✓ SlowCount
  - ❌ NormalCount
  - ⏱ InfiniteCount: error or timout after 30s; missing 7 requirement(s) for this product
- The hammer can complete at least 3 nails
  - ✓ SlowCount
- The hammer can complete at least 4 nails
  - ✓ NormalCount
- The hammer can finish a nail
  - ✓ Main
  - ✓ Overwork
  - ✓ Slow
  - ✓ Lazy
  - ✓ SlowLazy
  - ✓ SlowCount
  - ✓ NormalCount
  - ○ InfiniteCount: Not verified because there was an error or it timed out while checking another property.
- The hammer must complete a nail
  - ❌ Main

Figure 6: Output when running Uppex to verify all properties in the hammer project: to the prompt (left) and to the `report.html` file (right)

results from verifying each property. Each property is marked as passed, failed, or threw an error (e.g., time-out). Furthermore, a `report.html` file is created that clusters these results in a more useful form. The textual output and the HTML report of our hammer example can be found in Fig. 6.

4 Discussion

Our tool and methodology was first applied to our industrial use-case provided by the Alstom railway company on a signalling system, c.f. Section 2. Many of our design decisions were motivated by weekly discussions between academics and practitioners. Some of the insights gained by this collaboration using Uppex are summarised below.

- **Automata size:** The number of automata, locations, and variables easily increased when adding more details, reaching the 16 automata in Fig. 1. The high level of detail was appreciated by Alstom, as well as the use of variability to enable a precise verification of properties without needing to use statistical model checking approaches.

- **Non-determinism:** The high number of non-determinism resulting from allowing several actions to be taken in any order made it more difficult not only to verify, but also to predict the behaviour of the system. Consequently, a new version of this software is being prepared, with a finer scheduling control that reduces this non-determinism to a minimum.
- **Feature model:** The newly added structure to the features in the feature model contributed to a better understanding and insights of what can be modified in the formal model and how.
- **Attributes:** The possibility of using values in the `@Configurations` table facilitated the experimentation with different parameters without having to search through different tables for the values to update.
- **Optimal configurations:** The possibility of enriching `Uppex` to support the search for optimal configurations was considered, but this raised concerns regarding a possible increase of the learning curve to use `Uppex`. Introducing generic goal functions and cost values could compromise the ease of adoption of `Uppex`.
- **Feature model size:** As the feature model grows, the number of valid variants grows exponentially with the number of features. In `Uppex` this was not a concern, since it does neither generate all possible variants nor it attempts to find a variant that obeys some condition. Verifying if a single configuration is valid is computationally simple (linear on the size of the feature model). There is a risk of needing to manually add an increasingly large number of configurations to cover a relevant set of combinations, but we did not encounter this problem in our use-cases.

When compared the `Uppex` version used in our previous work with Alstom [16], the industrial partners mainly appreciated the possibility of providing numbers in the `@Configuration` table, avoiding the need to navigate through several other sheets. The added structure to the features brought from the feature model also contributed to a better understanding of what the features precisely captured (and resulted in some restructuring of features). Alstom developers were able to edit a shared Excel spreadsheet to adapt some configuration parameters, and were able to understand the generated `html` report, although the execution of `Uppex` with the model-checker was mainly carried by the academic partner. Furthermore, the usage of feature expressions in the `Features` column instead of single features also simplified our model, avoiding some previously added artificial features used to fine-tune the model.

5 Conclusion and future work

This paper reports on our recent attempt to include feature models represented in our configuring-spreadsheets in an intuitive way for developers, based on feature diagrams with integer attributes, and on how to exploit these for automatic analysis. This work was developed in collaboration with the Alstom railway company, within the VALU3S European project on verification and validation methods and tools.

Our experience showed that, on the one hand, it is useful to adapt the formal model and requirements by using a set of spreadsheets with key parameters. On the other hand it also highlighted that the pivotal notion of features was not yet fully exploited. This work includes support for feature models with attributes while preserving the simplicity of our spreadsheet-based interface, and keeping an easy-to-use solution that can be adopted by practitioners.

Based on the feedback from Alstom, possible ideas for future work include the following.

- **Coverage:** Currently it is possible to quickly grasp which configurations can validate each of the properties. However, it is hard to provide insights over how complete is this coverage, i.e., how much of the full system is validated for any given property. Counting the number of such configurations is a simple but not fully satisfactory approach. A better approach would be to *quantify the scope* of a configuration, e.g., how many locations can be reached, or which out of a set of reference reachability properties can be proven.
- **Other analysers:** We use Uppaal as our underlying model checker, but Uppex is general enough to be applied to other static analysis tools with little effort. For example, by using IMITATOR [1] instead we should be able to verify similar properties with a non-commercial tool and search for optimal parameters, and by using mCRL2 [7] instead we should be able to support the verification of properties focused on actions rather than states.
- **Deployment configurations:** The same configurations' table could be used to guide the customisation of deployment scripts, or other configuration files that can introduce the variability choices in the concrete software implementations.

Furthermore, we invite anyone in the community to submit suggestions or issues using GitHub's issue tracker system, or to contact us for future collaborations.

Acknowledgments

This work was supported by the CISTER Research Unit (UIDP/UIDB/04234/2020), financed by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology) and by project IBEX (PTDC/CCI-COM/4280/2021) financed by national funds through FCT. It is also a result of the work developed under projects and Route 25 (ref. TRB/2022/00061 – C645463824-00000063) funded by the EU/Next Generation, within the Recovery and Resilience Plan (RRP); and project VALU3S (ECSEL/0016/2019 – JU grant nr. 876852) financed by national funds through FCT and European funds through the EU ECSEL JU. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Sweden, Spain, Italy, France, Portugal, Ireland, Finland, Slovenia, Poland, Netherlands, Turkey – Disclaimer: This document reflects only the author's view and the Commission is not responsible for any use that may be made of the information it contains.

References

[1] Étienne André (2021): *IMITATOR 3: Synthesis of Timing Parameters Beyond Decidability*. In Alexandra Silva & K. Rustan M. Leino, editors: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, *LNCS* 12759, Springer, pp. 552–565, doi:10.1007/978-3-030-81685-8_26.

[2] Sven Apel, Don S. Batory, Christian Kästner & Gunter Saake (2013): *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, doi:10.1007/978-3-642-37521-7.

[3] Davide Basile, Maurice H. ter Beek, Alessio Ferrari & Axel Legay (2022): *Exploring the ERTMS/ETCS full moving block specification: an experience with formal methods*. *Int. J. Softw. Tools Technol. Transf.* 24(3), pp. 351–370, doi:10.1007/s10009-022-00653-3.

[4] Don S. Batory (2005): *A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite*. In Ralf Lämmel, João Saraiva & Joost Visser, editors: *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*, *Lecture Notes in Computer Science* 4143, Springer, pp. 3–35, doi:10.1007/11877028_1.

[5] Maurice H. ter Beek, Klaus Schmid & Holger Eichelberger (2019): *Textual variability modeling languages: an overview and considerations*. In Carlos Cetina, Oscar Díaz, Laurence Duchien, Marianne Huchard, Rick Rabiser, Camille Salinesi, Christoph Seidl, Xhevahire Tërnava, Leopoldo Teixeira, Thomas Thüm & Tewfik Ziadi, editors: *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume B, Paris, France, September 9-13, 2019*, ACM, pp. 82:1–82:7, doi:10.1145/3307630.3342398.

[6] Maurice H. ter Beek, Erik P. de Vink & Tim A. C. Willemse (2017): *Family-Based Model Checking with mCRL2*. In Marieke Huisman & Julia Rubin, editors: *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, *Lecture Notes in Computer Science* 10202, Springer, pp. 387–405, doi:10.1007/978-3-662-54494-5_23.

[7] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs & Tim A. C. Willemse (2019): *The mCRL2 Toolset for Analysing Concurrent Systems - Improvements in Expressivity and Usability*. In Tomás Vojnar & Lijun Zhang, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*, *LNCS* 11428, Springer, pp. 21–39, doi:10.1007/978-3-030-17465-1_-2.

[8] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay & Jean-François Raskin (2010): *Model checking lots of systems: efficient verification of temporal properties in software product lines*. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu & Sebastián Uchitel, editors: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, ACM, pp. 335–344, doi:10.1145/1806799.1806850.

[9] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis & Danny Bøgsted Poulsen (2015): *Uppaal SMC tutorial*. *International journal on software tools for technology transfer* 17, pp. 397–415, doi:10.1007/s10009-014-0361-y.

[10] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler & Klaus Schmid (2019): *Metrics for analyzing variability and its implementation in software product lines: A systematic literature review*. *Inf. Softw. Technol.* 106, pp. 1–30, doi:10.1016/j.infsof.2018.08.015.

[11] (2017): *Railway Applications. The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS). Generic RAMS Process*. Standard (N), CENELEC.

[12] (2020): *Railway applications. Communication, signalling and processing systems - Software for railway control and protection systems*. Standard (N), CENELEC.

[13] (2018): *Railway applications. Communication, signalling and processing systems. Safety related electronic systems for signalling*. Standard (N), CENELEC.

[14] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier & John Irwin (1997): *Aspect-oriented programming*. In: *ECOOP'97—Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11*, Springer, pp. 220–242, doi:10.1007/BFb0053381.

[15] Axel Legay, Anna Lukina, Louis-Marie Traonouez, Junxing Yang, Scott A. Smolka & Radu Grosu (2019): *Statistical Model Checking*. In Bernhard Steffen & Gerhard J. Woeginger, editors: *Computing and Software Science - State of the Art and Perspectives*, *Lecture Notes in Computer Science* 10000, Springer, pp. 478–504, doi:10.1007/978-3-319-91908-9_23.

[16] José Proença, Sina Borrami, Jorge Sanchez de Nova, David Pereira & Giann Spilere Nandi (2022): *Verification of Multiple Models of a Safety-Critical Motor Controller in Railway Systems*. In Simon Collart Dutilleul, Anne E. Haxthausen & Thierry Lecomte, editors: *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - 4th International Conference, RSSRail 2022, Paris, France, June 1-2, 2022, Proceedings*, *Lecture Notes in Computer Science* 13294, Springer, pp. 83–94, doi:10.1007/978-3-031-05814-1_6.

[17] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani & Nico Tanzarella (2010): *Delta-Oriented Programming of Software Product Lines*. In Jan Bosch & Jaejoon Lee, editors: *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*, *Lecture Notes in Computer Science* 6287, Springer, pp. 77–91, doi:10.1007/978-3-642-15579-6_6.

[18] Pierre-Yves Schobbens, Patrick Heymans & Jean-Christophe Trigaux (2006): *Feature Diagrams: A Survey and a Formal Semantics*. In: *14th IEEE International Conference on Requirements Engineering (RE 2006), 11-15 September 2006, Minneapolis/St.Paul, Minnesota, USA*, IEEE Computer Society, pp. 136–145, doi:10.1109/RE.2006.23.

[19] Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser & Thomas Thüm (2021): *Yet another textual variability language?: a community effort towards a unified language*. In Mohammad Reza Mousavi & Pierre-Yves Schobbens, editors: *SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021, Volume A*, ACM, pp. 136–147, doi:10.1145/3461001.3471145.

Serverless Scheduling Policies based on Cost Analysis

Giuseppe De Palma[1], Saverio Giallorenzo[1,2], Cosimo Laneve[1],
Jacopo Mauro[3], Matteo Trentin[1,3], Gianluigi Zavattaro[1,2]
[1]Università di Bologna, Italy
[2]Sophia Antipolis, INRIA, France
[3]University of Southern Denmark

Current proprietary and open-source serverless platforms follow opinionated, hardcoded scheduling policies to deploy the functions to be executed over the available workers. Such policies may decrease the performance and the security of the application due to locality issues (e.g., functions executed by workers far from the databases to be accessed). These limitations are partially overcome by the adoption of APP, a new platform-agnostic declarative language that allows serverless platforms to support multiple scheduling logics. Defining the "right" scheduling policy in APP is far from being a trivial task since it often requires rounds of refinement involving knowledge of the underlying infrastructure, guesswork, and empirical testing.

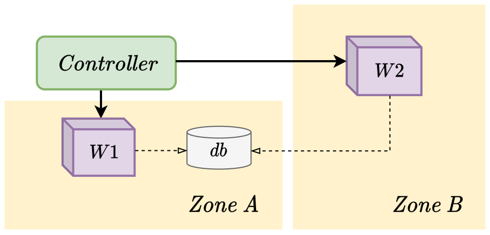
In this paper, we start investigating how information derived from static analysis could be incorporated into APP scheduling function policies to help users select the best-performing workers at function allocation. We substantiate our proposal by presenting a pipeline able to extract cost equations from functions' code, synthesising cost expressions through the usage of off-the-shelf solvers, and extending APP allocation policies to consider this information.

1 Introduction

Serverless is a cloud-based service that lets users deploy applications as compositions of stateless functions, with all system administration tasks delegated to the platform. Serverless has two main advantages for users: it saves them time by handling resource allocation, maintenance, and scaling, and it reduces costs by charging only for the resources used to perform work since users do not have to pay fur running idle servers [7]. Several managed serverless offerings are available from popular cloud providers like Amazon AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions, as well as open-source alternatives such as OpenWhisk, OpenFaaS, OpenLambda, and Fission. In all cases, the platform manages the allocation of function executions across available computing resources or workers, by adopting platform-dependent policies. However, the execution times of the functions are not independent of the workers since effects like *data locality* (the latencies to access data depending on the node position) can increase the run time of functions [6].

We visualise the issue by commenting on the minimal scenario drawn in Figure 1. There, we have two workers, W1 and W2, located in distinct geographical *Zones A* and *B*, respectively. Both workers can run functions that interact with a database (*db*) located in *Zone A*. When the function scheduler — the *Controller* — receives a request to execute a function, it must determine which worker to use. To minimise response time, the function scheduler must take into account the different computational capabilities of the workers, as well as their current workloads, and, for functions that interact with the database, the time to access the database. In the example, since W1 is geographically close to *db*, it can access *db* with lower latencies than W2.

APP [3, 2] is a declarative language recently introduced to support the *configuration of custom function-execution scheduling policies*. The APP snippet in Figure 1 codifies the (data) locality principle of the



```
- db_query:
  - workers:
    - wrk: W1
    - wrk: W2
    strategy: best_first
```

Figure 1: Example of function-execution scheduling problem and APP script.

example. Concretely, in the platform, we associate the functions that access *db* with a tag, called db_-query. Then, we include the scheduling rule in the snippet to specify that every function tagged db_query can run on either W1 or W2, and the strategy to follow when choosing between them is best_first, i.e., select the first worker in top-down order of appearance (hence giving priority to the worker W1 if available and not overloaded).

By featuring customised function scheduling policies, APP allows one to disentangle from platform-dependent allocation rules. This opens the problem of finding the most appropriate scheduling for serverless applications. The approach currently adopted by APP is to feature only a few generic well-established strategies, like the foregoing best_first. The policies are selected *manually*, when the APP script is written, based on the developer's insights on the behaviour of their functions.

In this paper, we propose the adoption of automatic procedures to define function scheduling policies based on information derived with a static analysis of the functions. Our approach relies on three main steps: (*i*) the definition of code analysis techniques for extracting meaningful scheduling information from function sources; (*ii*) the evaluation of scheduling information by a(n off-the-shelf) solver that returns cost expressions; (*iii*) the extension of APP to support allocation strategies depending on such expressions. In particular, we discuss the applicability of our approach on a minimal language for programming functions in serverless applications.

We start in Section 2 by defining our minimal language called miniSL (standing for mini Serverless Language) which includes constructs for specifying computation flow (via if and for constructs) and for service invocation (via a call construct). Then, by following [5, 8], we describe in Section 3 how to exploit a (behavioural) type system to automatically extract a set of equations from function source codes that define meaningful configuration costs. In Section 3 we also discuss how equations can be fed to off-the-shelf cost analyser (e.g., PUBS [1] or CoFloCo [4]) to compute cost expressions quantifying over-approximations of the considered configuration costs. These expressions are then used in Section 4 to define scheduling policies in an extension of APP, dubbed cAPP. Finally, in Section 5 we draw some concluding remarks.

2 The mini Serverless Language

The mini Serverless Language, shortened into miniSL, is a minimal calculus that we use to define the functions' behaviour in serverless computing. In particular, miniSL focuses only on core constructs to define operations to access services, conditional behaviour with simple guards, and iterations.

Function executions are triggered by events. At triggering time, a function receives a sequence of invocation parameters: for this reason, we assume a countable set of *parameter names*, ranged over by

p, p'. We also consider a countable set of *counters*, ranged over by i, j, used as indexes in iteration statements. Integer numbers are represented by n; service names are represented by h, g, $\cdots$. The syntax of miniSL is as follows (we use over-lines to denote sequences, e.g., p_1, p_2 could be an instance of $\overline{p}$):

$$\begin{array}{rcl}
\mathsf{F} & ::= & (\overline{p})\ \texttt{=>}\ \{\ \mathsf{S}\ \} \\
\mathsf{S} & ::= & \varepsilon \;\mid\; \texttt{call}\ \texttt{h}(\overline{\mathsf{E}})\ \mathsf{S} \;\mid\; \texttt{if}\ (\mathsf{G})\ \{\ \mathsf{S}\ \}\ \texttt{else}\ \{\ \mathsf{S}\ \} \;\mid\; \texttt{for}\ (i\ \texttt{in}\ \texttt{range}(\texttt{0},\mathsf{E}))\{\ \mathsf{S}\ \} \\
\mathsf{G} & ::= & \mathsf{E} \;\mid\; \texttt{call}\ \texttt{h}(\overline{\mathsf{E}}) \\
\mathsf{E} & ::= & n \;\mid\; i \;\mid\; p \;\mid\; \mathsf{E} \sharp \mathsf{E} \\
\sharp & ::= & \texttt{+} \;\mid\; \texttt{-} \;\mid\; \texttt{>} \;\mid\; \texttt{==} \;\mid\; \texttt{>=} \;\mid\; \texttt{\&\&} \;\mid\; \texttt{*} \;\mid\; \texttt{/}
\end{array}$$

A *function* F associates to a sequence of parameters $\overline{p}$ a statement S which is executed at every occurrence of the triggering event. *Statements* include the empty statement ε (which is always omitted when the statement is not empty); calls to external services by means of the call keyword; the conditional and iteration statements. The guard of a conditional statement could be either a boolean expression or a call to an external service which, in this case, is expected to return a boolean value. The language supports standard expressions in which it is possible to use integer numbers and counters. Notice that, in our simple language, the iteration statement considers an iteration variable ranging from 0 to the value of an expression E evaluated when the first iteration starts.

In the rest of the paper, we assume all programs to be well-formed so that all names are correctly used, i.e., counters are declared before they are used and when we use p, such p is an invocation parameter. Similarly, for each expression used in the range of an iteration construct, we assume that its evaluation generates an integer, and for each service invocation call h($\overline{E}$), we assume that h is a correct service name and $\overline{E}$ is a sequence of expressions generating correct values to be passed to that service. Calls to services include serverless invocations, which possibly execute on a different worker of the caller.

We illustrate miniSL by means of three examples. As a first example, consider the code in Listing 1 representing the call of a function that selects a functionality based on the characteristic of the invoker.

```
( isPremiumUser , par ) => {
  if( isPremiumUser ) {
    call PremiumService( par )
  } else {
    call BasicService( par )
  }
}
```

Listing 1: Function with a conditional statement guarded by an expression.

This code may invoke either a PremiumService or a BasicService depending on whether it has been triggered by a premium user or not. The parameter isPremiumUser is a value indicating whether the user is a premium member (when the value is true) or not (when the value is false). The other invocation parameter par must be forwarded to the invoked service. For the purposes of this paper, this example is relevant because if we want to reduce the latency of this function, the best node to schedule it could be the one that reduces the latency of the invocation of either the service PremiumService or the service BasicService, depending on whether isPremiumUser is true or false, respectively.

Consider now the following function where differently from the previous version, it is necessary to call an external service to decide whether we are serving a premium or a basic user.

```
( username , par ) => {
  if( call IsPremiumUser( username ) ) {
    call PremiumService( par )
  } else {
    call BasicService( par )
  }
}
```

Listing 2: Function with a conditional statement guarded by an invocation to external service.

Notice that, in this case, the first parameter carries an attribute of the user (its name) but it does not indicate (with a boolean value) whether it is a premium user or not. Instead, the necessary boolean value is returned by the external service `IsPremiumUser` that checks the username and returns true only if that username corresponds to that of a premium user. In this case, it is difficult to predict the best worker to execute such a function, because the branch that will be selected is not known at function scheduling time. If the user triggering the event is a premium member, the expected execution time of the function is the sum of the latencies of the service invocations of `IsPremiumUser` and `PremiumService` while, if the user is not a premium member, the expected execution time is the sum of the latencies of the services `IsPremiumUser` and `BasicService`. As an (over-)approximation of the expected delay, we could consider the worst execution time, i.e., the sum of the latency of the service `IsPremiumUser` plus the maximum between the latencies of the services `PremiumService` and `BasicService`. At scheduling time, we could select the best worker as the one giving the best guarantees in the worst case, e.g., the one with the best over-approximation.

Consider now a function triggering a sequence of map-reduce jobs.

```
( jobs , m, r ) => {
  for(i in range(0, m)) {
    call Map(jobs , i)
    for(j in range(0, r)) {
      call Reduce(jobs , i, j)
    }
  }
}
```

Listing 3: Function implementing a map-reduce logic.

The parameter `jobs` describes a sequence of map-reduce jobs. The number of jobs is indicated by the parameter `m`. The “map” phase, which generates `m` “reduce” subtasks, is implemented by an external service `Map` that receives the `jobs` and the specific index `i` of the job to be mapped. The “reduce” subtasks are implemented by an external service `Reduce` that receives the `jobs`, the specific index `i` of the job under execution, and the specific index `j` of the “reduce” subtask to be executed — for every `i`, there are `r` such subtasks. In this case, the expected latency of the entire function is given by the sum of `m` times the latency of the service `Map` and of `m` $\times$ `r` times the latency of the service `Reduce`. Given that such latency could be high, a user could be interested to run the function on a worker, only if the expected overall latency is below a given threshold.

3 The inference of cost expressions

In this section, we formalise how one can extract a cost program from miniSL code. Once extracted, we can feed this program to off-the-shelf tools, such as [4, 1], to calculate the cost expression of the related miniSL code.

Cost programs are lists of *equations* which are terms

$$f(\overline{x}) \;=\; \mathbb{e} + \sum_{i \in 0..n} f_i(\overline{\mathbb{e}_i}) \qquad\qquad [\,\varphi\,]$$

where variables occurring in the right-hand side and in φ are a subset of $\overline{x}$ and f and f_i are (cost) function symbols. Every function definition has a right-hand side consisting of

- a *Presburger arithmetic expression* $\mathbb{e}$ whose syntax is

  $$\mathbb{e} \;::=\quad x \;\;|\;\; q \;\;|\;\; \mathbb{e}+\mathbb{e} \;\;|\;\; \mathbb{e}-\mathbb{e} \;\;|\;\; q*\mathbb{e} \;\;|\;\; max(\mathbb{e}_1,\cdots,\mathbb{e}_k)$$

  where x is a variable and q is a positive rational number,
- a number of *cost function invocations* $f_i(\overline{\mathbb{e}_i})$ where $\overline{\mathbb{e}_i}$ are Presburger arithmetic expressions,
- the *Presburger guard* φ is a *linear conjunctive constraint*, *i.e.*, a conjunction of *constraints* of the form $\mathbb{e}_1 \geq \mathbb{e}_2$ or $\mathbb{e}_1 = \mathbb{e}_2$, where both $\mathbb{e}_1$ and $\mathbb{e}_2$ are Presburger arithmetic expressions.

The intended meaning of an equation $f(\overline{x}) \;=\; \mathbb{e} + \sum_{i\in 0..n} f_i(\overline{\mathbb{e}_i}) \;\; [\,\varphi\,]$ is that the cost of f is given by $\mathbb{e}$ and the costs of $f_i(\overline{\mathbb{e}_i})$, when the guard φ is true. Intuitively, $\mathbb{e}$ quantifies the specific cost of one execution of f without taking into account invocations of either auxiliary functions or recursive calls. Such additional cost is quantified by $\sum_{i\in 0..n} f_i(\overline{\mathbb{e}_i})$. The *solution of a cost program* is an expression, quantifying the cost of the function symbol in the first equation in the list, which is parametric in the formal parameters of the function symbol.

For example, the following cost program

$$\begin{array}{lcll} f(N,M) & = & M + f(N-1,M) & \quad [N \geq 1] \\ f(N,M) & = & 0 & \quad [N = 0] \end{array}$$

defines a function f that is invoked $N+1$ times and each invocation, excluding the last having cost 0, costs M. The solution of this cost program is the *cost expression* $N \times M$.

Our technique associates cost programs to miniSL functions by parsing the corresponding codes. In particular, we define a set of (inference) rules that gather fragments of cost programs that are then combined in a syntax-directed manner. As usual with syntax-directed rules, we use *environments* Γ, Γ', which are maps. In particular,

- Γ takes a service $\mathtt{h}$ or a parameter name p and returns a Presburger arithmetics expression, which is usually a variable. For example, if $\Gamma(\mathtt{h}) = X$, then X will appear in the cost expressions of miniSL functions using $\mathtt{h}$ and will represent the cost for accessing the service. As regards parameter names p, $\Gamma(p)$ represents values which are known at function scheduling time,
- Γ takes counters i and returns the type `Int`.

When we write $\Gamma + i : \mathtt{Int}$, we assume that i does not belong to the domain of Γ. Let C be a sum of cost of function invocations and let Q be a list of equations. Judgments have the shape

- $\Gamma \vdash \mathsf{E} : \mathbb{e}$, meaning that the value of the *integer expression* E in Γ is represented by (the Presburger arithmetic expression) $\mathbb{e}$,
- $\Gamma \vdash \mathsf{E} : \varphi$, meaning that the value of the *boolean expression* E in Γ is represented by (the Presburger guard) φ,
- $\Gamma \vdash \mathsf{S} : \mathbb{e} \;;\; \mathsf{C} \;;\; \mathsf{Q}$, meaning that the cost of S in the environment Γ is $\mathbb{e} + \mathsf{C}$ given a list Q of equations,

- $\Gamma \vdash \mathsf{F} : \mathsf{Q}$, meaning that the cost of a function F in the environment Γ is the list Q of equations.

We use the notation $var(\mathbb{e})$ to address the set of variables occurring in $\mathbb{e}$, which is extended to tuples $var(\mathbb{e}_1, \cdots, \mathbb{e}_n)$ with the standard meaning. Similarly $var(\sum_{i\in 0..n} f_i(\overline{\mathbb{e}_i}))$ is the union of the sets of variables $var(\overline{\mathbb{e}_0}), \cdots, var(\overline{\mathbb{e}_n})$.

The inference rules for miniSL are reported in Figure 2. They compute the cost of a program with respect to the calls to external services (whose cost is recorded in the environment Γ). Therefore, if a miniSL expression (or statement) has no service invocation, its cost is 0. Notice that in the rule [IF-EXP] we use the guard $[\,\neg\varphi\,]$, to model the negation of a linear conjunctive constraint φ, even if negation is not permitted in Presburger arithmetic. Actually, such notation is syntactic sugar defined as follows:

- let $\neg\varphi$ (the *negation* of a Presburger guard φ) be the *list* of Presburger guards

$$\begin{array}{rcl} \neg(\mathbb{e} \geq \mathbb{e}') & = & \mathbb{e}' \geq \mathbb{e}+1 \\ \neg(\mathbb{e} = \mathbb{e}') & = & \mathbb{e} \geq \mathbb{e}'+1 \ ; \ \mathbb{e}' \geq \mathbb{e}+1 \\ \neg(\mathbb{e} \wedge \mathbb{e}') & = & \neg\mathbb{e} \ ; \ \neg\mathbb{e}' \end{array}$$

  where ; is the list concatenation operator (the list represents a *disjunction of Presburger guards*),

- let $\neg\varphi = \varphi_1 \ ; \ \cdots \ ; \ \varphi_m$, where φ_i are Presburger guards, then

$$\Big(f(\overline{x}) \ = \ \mathbb{e} + \sum_{i\in 0..n} f_i(\overline{\mathbb{e}_i})\Big) \ [\neg\varphi] \quad \stackrel{\mathsf{def}}{=} \quad \Big\{ f(\overline{x}) \ = \ \mathbb{e} + \sum_{i\in 0..n} f_i(\overline{\mathbb{e}_i}) \quad [\varphi_j] \quad | \quad j \in 1..m \Big\} \ .$$

We now comment on the inference rules reported in Figure 2.[1]

Rule [CALL] manages invocation of services: the cost of call $\mathtt{h}(E)$ S is the cost of S plus the cost for accessing the service $\mathtt{h}$.

Rule [IF-EXP] defines the cost of conditionals when the guard is a Presburger arithmetic expression that can be evaluated at function scheduling time. We use a corresponding cost function, if_ℓ, whose name is fresh,[2] to indicate that the cost of the entire conditional statement is either the cost of the then-branch or the else-branch, depending on whether the guard is true or false. As discussed above, the use of the guard $\neg\varphi$ generates a list of equations.

Rule [IF-CALL] defines an upper bound of the cost of conditionals when the guard is an invocation to a service. At scheduling time it is not possible to determine whether the guard is true or false – *c.f.* the second example in Section 2. Therefore the cost of a conditional is the maximum between the cost $\mathbb{e}' + \mathsf{C}$ of the then-branch and the one $\mathbb{e}'' + \mathsf{C}'$ of the else-branch, plus the cost $\mathbb{e}$ to access to the service in the guard. However, considering that the expression $max(\mathbb{e} + \mathsf{C}, \mathbb{e}' + \mathsf{C}')$ is not a valid right-hand side for the equations in our cost programs, we take as over-approximation the expression $max(\mathbb{e}, \mathbb{e}') + \mathsf{C} + \mathsf{C}'$.

As regards iterations, according to [FOR], its cost is the invocation of the corresponding function, for_ℓ, whose name is fresh (we assume that iterations have pairwise different line-codes). The rule adds the counter i to Γ (please recall that $\Gamma + i : \mathtt{Int}$ entails that $i \notin dom(\Gamma)$). In particular, the counter i is the first formal parameter of for_ℓ; the other parameters are all the variables in $\mathbb{e}$, in notation $var(\mathbb{e})$ plus those in the invocations C (minus the i). There are two equations for every iteration: one is the case when i is out-of-range, hence the cost is 0, the other is when it is in range and the cost is the one of the body *plus* the cost of the recursive invocation of for_ℓ with i increased by 1.

The cost of a miniSL program is defined by [PRG]. This rule defines an equation for the function *main* and puts this equation as the first one in the list of equations.

[1]We omit rules for expressions E since they are straightforward: they simply return E if E is in Presburger arithmetics.

[2]We assume that conditionals have pairwise different line-codes and ℓ represents the line-code of the if in the source code.

[EPS]

$$\Gamma \vdash \varepsilon : 0 \,;\, \emptyset \,;\, \emptyset$$

[CALL]

$$\frac{\Gamma(\mathtt{h}) = \mathbb{e} \qquad \Gamma \vdash \mathsf{S} : \mathbb{e}' \,;\, \mathsf{C} \,;\, \mathsf{Q}}{\Gamma \vdash \mathtt{call}\ \mathtt{h}(\overline{\mathsf{E}})\ \mathsf{S} : \mathbb{e} + \mathbb{e}' \,;\, \mathsf{C} \,;\, \mathsf{Q}}$$

[IF-EXP]

$$\frac{\begin{array}{c} \Gamma \vdash \mathsf{E} : \varphi \qquad \Gamma \vdash \mathsf{S} : \mathbb{e}' \,;\, \mathsf{C} \,;\, \mathsf{Q} \qquad \Gamma \vdash \mathsf{S}' : \mathbb{e}'' \,;\, \mathsf{C}' \,;\, \mathsf{Q}' \qquad \mathit{if}_\ell \ \mathit{fresh} \\ \overline{w} = \mathit{var}(\mathbb{e}, \mathbb{e}', \mathbb{e}'') \cup \mathit{var}(\mathsf{C}, \mathsf{C}') \qquad \mathsf{Q}'' = \left[\begin{array}{ll} \mathit{if}_\ell(\overline{w}) = \mathbb{e}' + \mathsf{C} & [\, \varphi \,] \\ \mathit{if}_\ell(\overline{w}) = \mathbb{e}'' + \mathsf{C}' & [\neg\varphi] \end{array} \right] \end{array}}{\Gamma \vdash \mathtt{if}\ (\mathsf{E})\ \{\, \mathsf{S} \,\}\ \mathtt{else}\ \{\, \mathsf{S}' \,\} : 0 \,;\, \mathit{if}_\ell(\overline{w}) \,;\, \mathsf{Q}, \mathsf{Q}', \mathsf{Q}''}$$

[IF-CALL]

$$\frac{\Gamma(\mathtt{h}) = \mathbb{e} \qquad \Gamma \vdash \mathsf{S} : \mathbb{e}' \,;\, \mathsf{C} \,;\, \mathsf{Q} \qquad \Gamma \vdash \mathsf{S}' : \mathbb{e}'' \,;\, \mathsf{C}' \,;\, \mathsf{Q}'}{\Gamma \vdash \mathtt{if}\ (\mathtt{call}\ \mathtt{h}(\overline{\mathsf{E}}))\ \{\, \mathsf{S} \,\}\ \mathtt{else}\ \{\, \mathsf{S}' \,\} : \mathbb{e} + \mathit{max}(\mathbb{e}', \mathbb{e}'') \,;\, \mathsf{C} + \mathsf{C}' \,;\, \mathsf{Q}, \mathsf{Q}'}$$

[FOR]

$$\frac{\begin{array}{c} \Gamma \vdash \mathsf{E} : \mathbb{e} \qquad \Gamma + i : \mathsf{Int} \vdash \mathsf{S} : \mathbb{e}' \,;\, \mathsf{C} \,;\, \mathsf{Q} \qquad \overline{w} = (\mathit{var}(\mathbb{e}, \mathbb{e}') \cup \mathit{var}(\mathsf{C})) \setminus i \\ \mathit{for}_\ell \ \mathit{fresh} \qquad \mathsf{Q}' = \left[\begin{array}{ll} \mathit{for}_\ell(i, \overline{w}) = \mathbb{e}' + \mathsf{C} + \mathit{for}_\ell(i+1, \overline{w}) & [\, \mathbb{e} \geq i \,] \\ \mathit{for}_\ell(i, \overline{w}) = 0 & [\, i \geq \mathbb{e} + 1 \,] \end{array} \right] \end{array}}{\Gamma \vdash \mathtt{for}\ (i\ \mathtt{in}\ \mathtt{range}(0, \mathsf{E}))\{\, \mathsf{S} \,\} : 0 \,;\, \mathit{for}_\ell(0, \overline{w}) \,;\, \mathsf{Q}, \mathsf{Q}'}$$

[PRG]

$$\frac{\begin{array}{c} \Gamma \vdash \mathsf{S} : \mathbb{e} \,;\, \mathsf{C} \,;\, \mathsf{Q} \qquad \overline{w} = \mathit{var}(\overline{p}, \mathbb{e}) \cup \mathit{var}(\mathsf{C}) \\ \mathit{main}\ \mathit{fresh} \qquad \mathsf{Q}' = \mathit{main}(\overline{w}) = \mathbb{e} + \mathsf{C} \quad [\,] \end{array}}{\Gamma \vdash (\overline{p})\ \Rightarrow\ \{\, \mathsf{S} \,\} : \mathsf{Q}', \mathsf{Q}}$$

Figure 2: The rules for deriving cost expressions

As an example, in the following, we apply the rules of Figure 2 to the codes in Listings 1, 2 and 3. Let $\Gamma(\mathtt{isPremiumUser}) = u$, $\Gamma(\mathtt{PremiumService}) = P$ and $\Gamma(\mathtt{BasicService}) = B$. For Listing 1 we obtain the cost program

$$\begin{array}{rll} \mathit{main}(u, P, B) = & \mathit{if}_2(u, P, B) & [\,] \\ \mathit{if}_2(u, P, B) = & P & [\, u = 1 \,] \\ \mathit{if}_2(u, P, B) = & B & [\, u = 0 \,] \end{array}$$

For Listing 2, let $\Gamma(\mathtt{IsPremiumUser}) = K$. Then the rules of Figure 2 return the single equation

$$\mathit{main}(K, P, B) = \quad K + \mathit{max}(P, B) \qquad [\,]$$

For 3, when $\Gamma(\mathtt{m}) = m$, $\Gamma(\mathtt{r}) = r$, $\Gamma(\mathtt{Map}) = M$ and $\Gamma(\mathtt{Reduce}) = R$, the cost program is

$$\begin{array}{rll} \mathit{main}(m, r, M, R) = & \mathit{for}_2(0, m, r, M, R) & [\,] \\ \mathit{for}_2(i, m, r, M, R) = & M + \mathit{for}_4(0, r, R) + \mathit{for}_2(i+1, m, r, M, R) & [\, m \geq i \,] \\ \mathit{for}_2(i, m, r, M, R) = & 0 & [\, i \geq m + 1 \,] \\ \mathit{for}_4(j, r, R) = & R + \mathit{for}_4(j+1, r, R) & [\, r \geq j \,] \\ \mathit{for}_4(j, r, R) = & 0 & [\, j \geq r + 1 \,] \end{array}$$

The foregoing cost programs can be fed to automatic solvers such as Pubs [1] and CoFloCo [4]. The evaluation of the cost program for Listing 1 returns $\mathit{max}(P, B)$ because u is unknown. On the contrary, if u is known, it is possible to obtain a more precise evaluation from the solver: if $u = 1$ it is possible to ask the solver to consider $main(1,P,B)$ and the solution will be P, while if $u = 0$ it is possible to ask the solver to consider $main(0,P,B)$ and the solution will be B. The evaluation of $main(K,P,B)$ for Listing 2 gives the expression $K + max(P,B)$, which is exactly what is written in the equation. This is reasonable because, statically, we are not aware of the value returned by the invocation of `IsPremiumService`. Last, the evaluation of the cost program for Listing 3 returns the expression $m \times (M + r \times R)$.

4 From APP to cAPP

We now discuss the extension of APP that we plan to realise, where function scheduling policies could depend on the costs associated with the possible execution of the functions on the available workers.

Before discussing the extensions towards cAPP, we briefly introduce the APP syntax and constructs, reported in Figure 3, as found in its first incarnation by De Palma et al. [3]

The APP Language

An APP script is a collection of tagged scheduling policies. The main, mandatory component of any policy (identified by a *policy_tag*) are the `workers` therein, i.e., a collection of labels that identify on which workers the scheduler can allocate the function. The assumption is that the environment running APP establishes a 1-to-1 association so that each worker has a unique, identifying label. A policy, associate to every function a list of one or more *block*s, each including the `worker` clause to state on which workers the function can be scheduled and two optional parameters: the scheduling `strategy`, followed to select one of the workers of the block, and an `invalidate` condition, which determines when a worker cannot host a function. When a selected worker is invalid, the scheduler tries to allocate the function on the rest of the available workers in the block. If none of the workers of a block is available, then the next block is tried. The last clause, `followup`, encompasses a whole policy and defines what to do when no *block*s of the policy managed to allocate the function. When set to `fail`, the scheduling of the function fails; when set to `default`, the scheduling continues by following the (special) `default` policy.

As far as the `strategy` is concerned, it allows the following values: `platform` that applies the default selection strategy of the serverless platform; `random` that allocates functions stochastically among the workers of the block following a uniform distribution; `best-first` that allocates functions on workers based on their top-down order of appearance in the block. The options for the `invalidate` are instead: `overload` that invalidates a worker based on the default invalidation control of the platform; `capacity_used` that invalidates a worker if it uses more than a given percentage threshold of memory; `max_concurrent_invocations` that invalidates a worker if a given number of function invocations are already currently executed on the worker.

Towards cAPP

Our proposal to extend APP to handle cost-aware scheduling policies entails two major modifications: (*i*) extending the APP language to express cost-aware scheduling policies, (*ii*) implementing a new controller that selects the correct worker following the cost-aware policies.

As far as (i) is concerned, we discuss at least two relevant ways in which costs can be used. The first one is a new selection strategy named `min_latency`. Such a strategy selects, among some available workers, the one which minimises a given cost expression. The second one is a new invalidation condition named `max_latency`. Such a condition invalidates a worker in case the corresponding cost expression is greater than a given threshold.

$$
\begin{array}{lcl}
\textit{policy\_tag} & \in & \textit{Identifiers} \cup \{\texttt{default}\} \qquad \textit{worker\_label} \in \textit{Identifiers} \qquad n \in \mathbb{N} \\
\textit{app} & ::= & \overline{\textit{tag}} \\
\textit{tag} & ::= & \textit{policy\_tag} \texttt{ : } \overline{\texttt{-}\ \textit{block}}\ \textit{followup}? \\
\textit{block} & ::= & \texttt{workers: } [\ \texttt{*} \mid \overline{\texttt{-}\ \textit{worker\_label}}\ ] \\
 & & (\texttt{strategy: } [\ \texttt{random} \mid \texttt{platform} \mid \texttt{best\_first}\ ])? \\
 & & (\texttt{invalidate: } [\ \texttt{capacity\_used : } n\% \\
 & & \qquad\qquad\quad \mid \texttt{max\_concurrent\_invocations: } n \\
 & & \qquad\qquad\quad \mid \texttt{overload}\ ])? \\
\textit{followup} & ::= & \texttt{followup: } [\ \texttt{default} \mid \texttt{fail}\ ]
\end{array}
$$

Figure 3: The APP syntax.

We dub cAPP the cost-aware extension of APP and illustrate its main features by showing examples of cAPP scripts that target the functions in Listings 1–3.

```
- premUser:
 - workers:
     - wrk: W1
     - wrk: W2
   strategy: min_latency
```

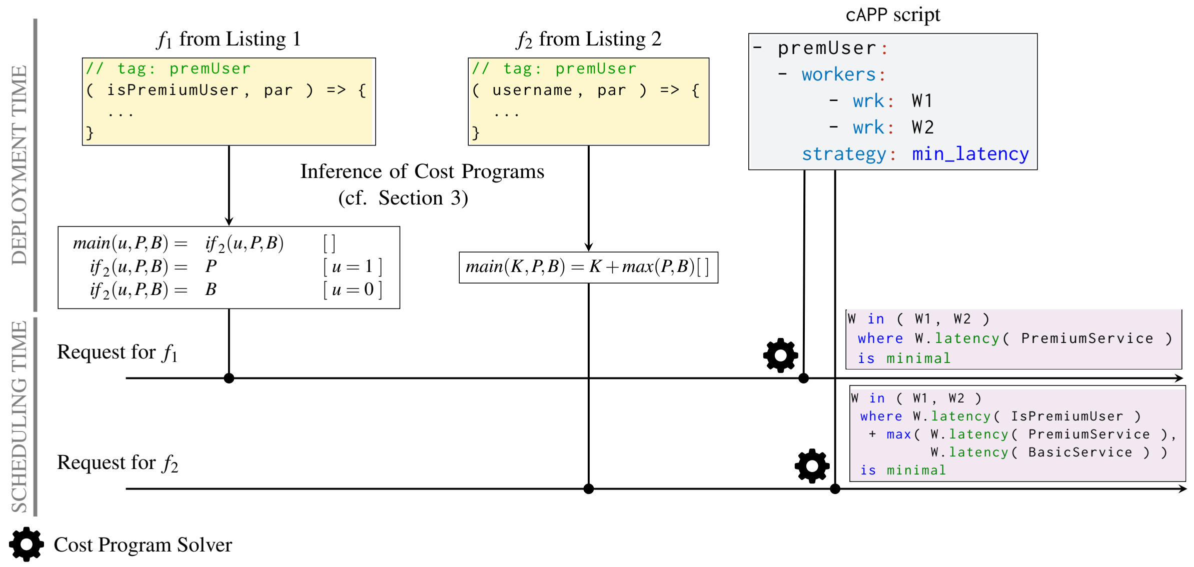
Listing 4: cAPP script for Listings 1 and 2.

```
- mapReduce :
  - workers:
    - wrk: W1
    - wrk: W2
  strategy: random
  invalidate:
    max_latency: 300
```

Listing 5: cAPP script for Listing 3.

In Listing 4, we define a cAPP script where we assume to associate the tag `premUser` to both the functions at Listing 1 and 2. In the script, we specify to follow the logic `min_latency` to select among the two workers, `W1` and `W2` listed in the `workers` clause, and prioritises the one for which the solution of the cost expression is minimal.

To better illustrate the phases of the `min_latency` strategy, we depict in Figure 4 the flow, from the deployment of the cAPP script to the scheduling of the functions in Listings 1 and 2. When the cAPP script is created, the association between the functions code and their cAPP script is specified by tagging the two functions with `//tag:premUser`. In this phase, assuming the scheduling policy of the cAPP script requires the computation of the functions cost, the code of the functions is used to infer the corresponding cost programs. When the functions are invoked, i.e., at scheduling time, we can compute the solution of the cost program, given the knowledge of the invocation parameters. For instance, for the function in Listings 1, it is possible to invoke the solver with either $main(1,P,B)$ or $main(0,P,B)$ depending on the actual invocation parameter. Figure 4 illustrates this last part with the horizontal "request" lines found at the bottom. In particular, when we receive a request for the function at Listing 1, we take its cost program (represented by the intersection point on the left) and its corresponding cAPP policy to implement the expected scheduling policy. We can implement this behaviour in two steps. First, the solver solves the cost programs (depicted by the gear); then, we compute the obtained cost expression for each of the possible workers (in this case, `W1` and `W2`) by instantiating the parameter representing the cost of invocation of the external services, with an estimation of the latencies from the considered workers. In this case, given the `min_latency` strategy, the worker that minimises the `latency` to contact `PremiumService` will be selected. This last step regards the second point (*ii*) mentioned at the beginning of this section, i.e., the



```
// tag: premUser
( isPremiumUser, par ) => {
  ...
}
```

```
// tag: premUser
( username, par ) => {
  ...
}
```

```
- premUser:
  - workers:
      - wrk: W1
      - wrk: W2
    strategy: min_latency
```

$$
\begin{array}{lll}
main(u,P,B) = & if_2(u,P,B) & [\,] \\
if_2(u,P,B) = & P & [\,u=1\,] \\
if_2(u,P,B) = & B & [\,u=0\,]
\end{array}
$$

$$main(K,P,B) = K + max(P,B)[\,]$$

```
W in ( W1, W2 )
 where W.latency( PremiumService )
 is minimal
```

```
W in ( W1, W2 )
 where W.latency( IsPremiumUser )
  + max( W.latency( PremiumService ),
         W.latency( BasicService ) )
 is minimal
```

Figure 4: Flow followed, from deployment to scheduling, of the functions at Listings 1 and 2.

modifications we need to perform on the controller to let it execute the newly introduced cost-aware strategies at scheduling time.

For `max_latency`, once a worker is selected using a given strategy, its corresponding cost is computed in order to check whether the selection is invalid (i.e., if we can consider the worker able to execute the function, given the invalidation constraints of the script). To illustrate this second occurrence, we look at the cAPP code we wrote for the map-reduce function in Listing 5, and we illustrate it using Figure 5. As seen above, we start (top-most box) from the deployment phase, where we tag the function (`//tag: mapReduce`) and we proceed to compute its cost program, obtaining the associated cost expression. Then, when we receive a request for that function, we trigger the execution of the cAPP policy, which selects one of the two workers `W1` or `W2` at `random` and checks their validity following the logic shown at the bottom of Figure 5, i.e., we solve the cost program and then compute the corresponding cost expression by replacing the parameters `m` and `r` with the `latency` to contact the `Map` and `Reduce` services from the selected worker, and possibly invalidate it if the computed value is greater than `300`. In the function's code, for simplicity, we abstract away the coordination logic between `Map` and `Reduce` (which usually performs a multipoint scatter-gather behaviour) by offloading it to external services (e.g., a database contacted by the functions).

These new `strategy` and `invalidate` parameters added for cAPP interact with the cost-inference logic presented in Section 3. As shown in Figure 4, the definition of the `strategy` and `invalidate` parameters, as well as the cost inference, happen independently, when the cAPP script is deployed. A strategy indeed (e.g., `min_latency`) is not tied to any specific cost expression. For example, the user can define the `premUser` policy (see the cAPP script on the right-hand side of Figure 4) before having deployed any function with that tag. When functions are deployed on the platform (centre and left-hand side of Figure 4), the cAPP runtime performs the inference of programs' costs. When instead a request for the execution of a function reaches the platform, the cAPP use the cost expressions and create the logic of selection/invalidation down to its runtime form. For instance, in Figure 4, the scheduling of function f_1 compiles the `min_latency` logic using the reduced form P (the cost of accessing service `PremiumService`) since at scheduling time the parameter `isPremiumUser` (represented by the variable u

```
// tag: mapReduce
( jobs, m, r ) => {
  for(i in range(0, m)) {
    call Map(jobs, i)
    for(j in range(0, r)) {
      call Reduce(jobs, i, j)
    }
  }
}
```

$$\Downarrow$$

$$\begin{array}{rll}
main(m,r,M,R) = & for_2(0,m,r,M,R) & [\,] \\
for_2(i,m,r,M,R) = & M + for_4(0,r,R) + for_2(i+1,m,r,M,R) & [\,m \geq i\,] \\
for_2(i,m,r,M,R) = & 0 & [\,i \geq m+1\,] \\
for_4(j,r,R) = & R + for_4(j+1,r,R) & [\,r \geq j\,] \\
for_4(j,r,R) = & 0 & [\,j \geq r+1\,]
\end{array}$$

$$\Downarrow$$

```
Cost Expression: m*(M + r*R)
```

$$\Downarrow$$

```
W in ( W1, W2 )
 where m *( W.latency( Map ) + r * W.latency( Reduce ) )
 is < 300
```

Figure 5: The map-reduce function, its cost analysis, and scheduling invalidation logic.

in the related cost equations in Figure 4) is known, which in the example we value to 1 (i.e., the request is from a premium user). From the reduced cost expression we can obtain the cAPP selection logic on the right-hand side of Figure 4: select that worker, among the one provided in the cAPP block, that minimises (`is minimal`) the `latency` of interaction with the `PremiumService` service.

For completeness, we can draw a parallel example for the invalidation parameter by looking at Figure 5. There, once we have a request for the map-reduce function, we take the cost expression calculated at deployment time, whose (integer) values represented by `m` are `r` are known at scheduling time, and we compile the `invalidate` logic, `max_latency:300` — for the map-reduce function, the logic declares invalid any worker whose cost `m *( W.latency( Map ) + r * W.latency( Reduce ) )` exceeds the set `300` threshold.

5 Conclusion

We have presented a proposal for an extension of the APP language, called cAPP, to make function scheduling cost-aware. Concretely, the extension adds new syntactic fragments to APP so that programmers can govern the scheduling of functions towards those execution nodes that minimise their calculated latency (e.g., increasing serverless function performance) and avoids running functions on nodes whose execution time would exceed a maximal response time defined by the user (e.g., enforcing quality-of-service constraints). The main technical insights behind the extension include the usage of inference rules to extract cost equations from the source code of the deployed functions and exploiting dedicated solvers to compute the cost of executing a function, given its code and input parameters.

Growing our proposal into a usable APP extension is manyfold. The cost inference of Section 2 programs is under active development at the time of writing.[3] While the solution of the cost equations can be done by off-the-shelf tools (e.g.,CoFloCo [4]), another important component to develop is the cAPP runtime to generate cAPP rules from the cost equations when functions are scheduled and interact with the workers available in the platform to collect the measures that characterise the costs sustained by the workers (e.g., the latency endured by a worker when contacting a given service).

Implementing the cAPP runtime and proving the feasibility of cost-aware function scheduling is only the first move along the way. Indeed, in Section 4 (illustrated in Figure 4) we described a naïve approach where we solve the cost equations of an invoked function at scheduling time, but this computation step could delay the scheduling of the function. This challenge calls for further investigation. On the one hand, we shall investigate if the problem presents itself in practice, i.e., if developers would actually write functions whose cost equations take too much time for the available engines to solve. On the other hand, we envision working on models and techniques that can make the problem treatable (e.g., via heuristics and over-approximations), possibly complementing the former with architectural solutions, like the inclusion of caching systems that allows us to compute the actual cost of function invocations once and timeouts paired with sensible default strategies which would keep the system responsive.

Acknowledgement

Research partly supported by the H2020-MSCA-RISE project ID 778233 “Behavioural Application Program Interfaces (BEHAPI)” and by the SERICS project (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

[1] Elvira Albert, Puri Arenas, Samir Genaim & Germán Puebla (2008): *Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis*. In María Alpuente & Germán Vidal, editors: *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*, *Lecture Notes in Computer Science* 5079, Springer, pp. 221–237, doi:10.1007/978-3-540-69166-2_15. Available at https://doi.org/10.1007/978-3-540-69166-2_15.

[2] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, Matteo Trentin & Gianluigi Zavattaro (2022): *A Declarative Approach to Topology-Aware Serverless Function-Execution Scheduling*. In Claudio Agostino Ardagna, Nimanthi L. Atukorala, Boualem Benatallah, Athman Bouguettaya, Fabio Casati, Carl K. Chang, Rong N. Chang, Ernesto Damiani, Chirine Ghedira Guegan, Robert Ward, Fatos Xhafa, Xiaofei Xu & Jia Zhang, editors: *IEEE International Conference on Web Services, ICWS 2022, Barcelona, Spain, July 10-16, 2022*, IEEE, pp. 337–342, doi:10.1109/ICWS55610.2022.00056.

[3] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro & Gianluigi Zavattaro (2020): *Allocation Priority Policies for Serverless Function-Execution Scheduling Optimisation*. In Eleanna Kafeza, Boualem Benatallah, Fabio Martinelli, Hakim Hacid, Athman Bouguettaya & Hamid Motahari, editors: *Service-Oriented Computing - 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14-17, 2020, Proceedings*, *Lecture Notes in Computer Science* 12571, Springer, pp. 416–430, doi:10.1007/978-3-030-65310-1_29.

[4] Antonio Flores-Montoya & Reiner Hähnle (2014): *Resource Analysis of Complex Programs with Cost Equations*. In Jacques Garrigue, editor: *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, *Lecture Notes in Computer Science* 8858, Springer, pp. 275–295, doi:10.1007/978-3-319-12736-1_15. Available at https://doi.org/10.1007/978-3-319-12736-1_15.

[3] https://github.com/minosse99/CostCompiler

[5] Abel Garcia, Cosimo Laneve & Michael Lienhardt (2017): *Static analysis of cloud elasticity*. *Sci. Comput. Program.* 147, pp. 27–53, doi:10.1016/j.scico.2017.03.008. Available at `https://doi.org/10.1016/j.scico.2017.03.008`.

[6] Scott Hendrickson, Stephen Sturdevant, Edward Oakes, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau & Remzi H. Arpaci-Dusseau (2016): *Serverless Computation with OpenLambda*. 41. Available at `https://www.usenix.org/publications/login/winter2016/hendrickson`.

[7] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica & David A. Patterson (2019): *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley.

[8] Cosimo Laneve & Claudio Sacerdoti Coen (2021): *Analysis of smart contracts balances*. *Blockchain: Research and Applications* 2(3), p. 100020, doi:https://doi.org/10.1016/j.bcra.2021.100020. Available at `https://www.sciencedirect.com/science/article/pii/S2096720921000154`.

Product Line Management with Graphical MBSE Views

Pascal Krapf
Syscience
Villebon sur Yvette, France
pascal.krapf@syscience.fr

Sébastien Berthier
Syscience
Villebon sur Yvette, France
sebastien.berthier@syscience.fr

Nicole Levy
CEDRIC-CNAM
Paris, France
nicole.levy@cnam.fr

Abstract : Reducing the cost and delay and improving quality are major issues for product and software development, especially in the automotive domain. Product line engineering is a well-known approach to engineer systems with the aim to reduce costs and development time as well as to improve the product quality. Feature models enable to make logical selection of features and obtain a filtered set of assets that compose the product. We propose to use a color code in feature models to make possible decisions visual in the feature tree. The color code is explained and its use is illustrated. The completeness of the approach is discussed.

Keywords : Configuration, variants, product line, model-based system engineering (MBSE)

1 Introduction

Reducing the cost and delay and improving quality are major issues for product and software development. To achieve these challenges, strategies for reuse and standardization of products and software have been developed. In this way, development and validation of components and software assets are mutualitized over several projects, which reduces the global cost of the product. Indeed, the number of individual assets that are required to build complex products like personal cars, aircrafts, trains or industrial facilities can reach several thousands (from ten to several hundred thousand). Moreover only some of them are present in all products, while the others are associated to particular products. Products can be differentiated by several characteristics:

- Products may differ in the offered functionalities.
- Products may differ in performance values.
- Products may differ in the non-functional properties.
- Products may differ in the chosen execution platform.

In the automotive domain, a widespread practice is to define from the beginning a Product Line (PL) approach [22]. It consists in designing a set of defined products embedding physical and software components developed from a common set of core assets and having a managed set of variable features [11].

Developing from the beginning a PL, means to focus on the variability and on the potential differences between products. The method we applied focuses on the creation of a product line right from the initial product development stage. The aim is to propose possible variants from the very start, knowing that some others could be added to later. It's a very different approach to parameterization. Defining a parameterized product means concentrating on the common functionalities. The overall architecture is generally not variable and, as a result, non-functional properties are less variable and less emphasized.

Variant management languages and associated tools have been developed with high expressiveness to describe product lines [15, 9, 14, 17, 5]. However, up to now, their deployment in the automotive industry is not effective.

Maurice H. ter Beek and Clemens Dubslaff (Eds.):
Proceedings of the First Workshop on Trends
in Configurable Systems Analysis (TiCSA'23)
EPTCS 392, 2023, pp. 53–65, doi:10.4204/EPTCS.392.4

In the automotive industry, dozens of development teams are specialized in various domains of engineering. They all contribute to the definition product variants. They each have specific concerns about variability and favor processes and tools suited to their specific concern. But at the end of development, a reduced number of people in the project team has to be able to select the project variants without being expert in all the engineering domains. Thus, the way the variability is structured has to be understandable by people outside the specific engineering domain.

This is why powerful specialized tools are of little help when deploying a PL approach on an industrial project in the automotive domain.

Our proposal is to define processes methods and an associated tool with simple and visual interfaces made intuitive for users not accustomed to software-oriented tools. Even if less powerful for constraints expression, consistence analysis and solving capabilities than existing ones, such a tool may be better suited to meet user acceptance in our specific domain.

The first section is the present introduction.

In the second section of this article, we list a set of qualities targeted when configuring a system.

In the third section, we present useful concepts issued from our experience that are frequently used in industries managing different kinds of product lines.

In the fourth section, we present our approach to build and use PL inspired from FODA's feature models [16]. We use for this purpose our existing system engineering modeling platform, which is under improvement.

In the fifth section, we discuss the possibility to handle any kind of logical constraints in the framework we have defined.

2 Overall strategy and targeted properties

2.1 Parameterized software versus Product Line

The most straightforward way to develop a set of related products is to develop a first product and adapt this product using tune-able parameters and adding components.

The activity of making a system tune-able can be split into:

- identify the possible adjustable parameters and components associated to variabilities,
- define the values to be selected for the parameters and design specific additional components.

Such strategies have been applied by carmakers and have shown some limitations:

- Trade-offs are driven by the first designed product, which does not mean global optimization for all products.
- Adding variabilities as add-ons considerably increase the complexity of the product and can increase the risk of malfunction.
- It generally requires deep knowledge of the product to be able to tune the variable parameters, while companies often want the configuration to be done by non-specialists.

As a consequence, carmakers are preferring the product line approach [24]. A set of defined products that share a common, managed set of features and are developed from a common set of core assets [11].

2.2 Qualities targeted when configuring a system

Following system engineering practices, the first step is to capture needs about the variant configuration management framework. The need is a framework (process and tool) that has the following characteristics [10]:

- **Operability**: the number of actions to select a product variant is reduced and available to human decision, both for the first setup and for later updates,
- **Evolutivity**: it is possible to add features and parts to the product line and continue to use former versions. When the product line is enriched, new features are added along with new added constraints,
- **Reusability**: parts and groups of parts can be reused with confidence without modification in new products or new product lines,
- **Simplicity**: no deep knowledge about the system design is necessary to select a variant. Parameterization can be done in a way that is accessible for non-specialists of the domain,
- **Modularity**: Architects can select coherent subsets of the product line by the selection of sets of variants,
- **Consistency**: Compliance with design rules constraining the choice of variants is ensured. These design rules can be of a norm, regulations or chosen method to be followed.

A product line allows to abstract the construction from the configuration of the reusable components, to identify reasoning and decisions behind the selection of a configuration. Reasoning means building a series of relations between causes and consequences while taking into account a set of logical constraints. A PL management framework is likely to satisfy the former list of characteristics.

3 Field data

3.1 Architectural point of views

Cyber-physical systems are more and more developed using Model based system engineering (MBSE). In these models, systems and their relations with their environment are described by views corresponding to different viewpoints. The approach we were using includes the following viewpoints [2]:

- **Operational viewpoint**, focused on the concern of how the system is operated and interacts with surrounding systems.
- **Functional viewpoint** focused on the concern of system functionalities, functional interfaces, functioning modes and behavior.
- **Organic viewpoint** focused on the concern of system components, components allocation of functions and requirements and how internal components interact.

These models are used for the product development and are included in the digital twin of the product [19]. The following paragraphs describe some model elements that are of interest for variant management. We empathize that variant management is about selecting components (organic viewpoint), but with key drivers coming from the other viewpoints.

3.2 Operational variability

A use case is a specific situation in which a product could potentially be used. For example personal cars can be driven on railways, on open roads, in town, on tracks. They can be parked on road, in garden lane or in a garage, etc. Each use case carries specific requirements the car has to comply with in order to satisfy the customer. Since all customers do not have the same expectations and use cases, different variants of cars are commercialized. Level of outfitting, seat comfort, acoustics, dynamism, speed, smoothness, product durability in specific mission profiles are operational characteristics that can be in the scope of variant management.

3.3 Functional variability

Customers can choose among a set of functionalities for their personal car. For example, driver assistance systems, guiding assistance, comfort adaptation, entertainment for passengers, door opening etc. The are an important source of variability. Some of them induce the presence of specific components like sensors and actuators, but others can be activated or deactivated by software. End to end functionalities are split into sub-functions forming functional chains. Each sub-function uses inputs to produce some outputs. The combination of these functions ensure the overall product functionalities.

3.4 Component variability

3.4.1 Bill of material

In industry, the list of all parts or assets that can be purchased to build a product in a product line must be managed. This list is called the "Bill Of Material", or BOM. A 150% BOM is a list containing the parts to be used to produce the whole set of products of the product line. An individual product will not include all the parts, but only a subset of them: the BOM containing only the parts for an individual product is called 100% BOM. The BOM is managed as a list in which each standardized part appears only once (an eventual multiplicity will be managed later as we will see).

A BOM may contain up to several hundreds of individual parts, and a large proportion of these parts (frequently 10% to 50%) are linked to a variant, as they are not present in all individual products. The first target of variability management is to select efficiently parts corresponding to a specific product. If one wants to select individually each part, one would have to know all the components that are required for each functionality. This choice tends to be impossible for functionalities requiring several hundred thousand parts. Even for engineers in the appropriate field, this is just impossible.

3.4.2 Asset library

Complex systems are often software intensive, meaning that functions are realized by software. Software intensive systems are made of a combination of many interacting software components. Thus, the issue of variant management for physical parts is mirrored in the software domain. Software components are listed in a software library. Assets also include models, specifications, assembly instructions, procedures, tools, validation facilities, safety assessments etc. Actually, any asset contributing to the product definition can be in the scope of variant management. Thus, physical or purchasable parts listed in a BOM are not enough to define all possibilities to build a product line. It is more flexible and more accurate to consider an asset library that contains any kind of artifacts.

3.4.3 Product breakdown structure

System engineering is the general framework used to develop complex products [1, 3, 2, 4, 20]. A product is broken down into systems. Each system is broken down into subsystems, and so on until reaching individual parts that can be subcontracted and purchased from suppliers. Assets are organized in a tree structure called Product Breakdown Structure (PBS). The PBS contains components that have an active role in the system functioning (sensors, control unit...) as well as components associated to liabilities (tight box, firewall...). Software engineers generally build their software with software components. The software components library is a part of the PBS. Thus, the PBS gives a structured view of the assets that constitute the product.

Engineering teams organize the PBS according to the system breakdown. This breakdown often reflects the organization of the engineering teams and corresponding engineering domains. The manufacturing team has interest to organize the PBS according to how the system is assembled. This may not fit with the engineering team's organization. The purchasing team may want to structure the PBS according to possible suppliers. The maintenance team may want to organize the PBS in accordance with maintenance schedule and process. If several teams like purchasing, manufacturing, maintenance, use the same PBS then, the structure has to be a compromise between their needs. Companies that try to manage variants by merging all parts in a single PBS that is managed in an Enterprise Resource Planning (ERP) tool often create dissatisfaction in every domain team. This may contribute to the high failure rate of ERP deployment projects [12].

3.5 Feature model

The word "feature" refers to a characteristic or a set of characteristics of the product line. As already discussed, PBS is not the only model element that is impacted by variant management. Thus, there is no reason to have a one-to-one correspondence of nodes of the PBS and features. The PBS structure does not necessarily reflect a selection logic of the components. Requirement satisfaction may involve the contribution of several different parts located in different branches of the PBS.

General software qualities like cybersecurity, energy consumption efficiency, human machine interfaces, etc. often must be managed with variability. The random selection of software components does not ensure the quality of the final product.

Companies describe the features that are likely to be variant in a feature model. It describes variable features that can be selected for an individual product within the product line.

Feature Diagrams (FD) are a family of modeling languages used to address the description of products in product lines [23]. FD were first introduced by Kang as part of the FODA (Feature Oriented Domain Analysis) method back in 1990 [16].

Feature models are generally represented in a tree structure. Each node is a feature that can be selected. A natural rule is to select a son node only if the father node is already selected. In any product line, the feature choice is submitted to rules allowing or forbidding some associations. Rules can result from physics (not enough place), regulation (no such combination of functions), marketing, etc. For example, cars can have diesel, gasoline, hybrid or electrical engines, but only one among this list. These features are not independent, and the dependence is only partially represented by the position in the feature tree. When complex constraints are involved, then it often requires a solver to be able to determine whether a set of selected features comply with these rules. The problem of deciding if a set of logical sentences has possible solutions is known to be NP complex [23].

Variable characteristics usually reach the number of several hundreds to several thousand for cars or aircrafts. Thus, it is still difficult to make choices because of the need to be coherent. Furthermore, the number of possible configurations is still enormous. If 100 nodes can be selected in a feature tree, then the number of possible different products is equal to 2 to the power of 100. This number of combinations cannot be managed extensively. A structured methodology with associated tools is needed. The domain engineer designs the product line in a way to minimize circular or interwoven constraints. The aim is to make the feature model easily understandable to applications engineers. The PL engineer has to define a smart structure for the product line, and the application engineer needs deep knowledge of the product line to select features without losing time with attempts and errors.

3.6 Variation criteria

The PL is described in a model, that contains products assets. Some assets are present in all individual products. These assets form the invariant backbone of the PL. The other assets are present or absent depending on the features that are chosen. A variation criterion is a logical formula, that defines the asset variability. This logical formula is expressed using the features of the feature model. The asset is present in the product if the formula is evaluated TRUE. In this way, assets can be filtered according to the feature selection. The completeness and coherency of this association between assets and features fully relies on the PL design engineer.

4 Framework for variant management

In order to be efficient, companies need a framework of combined and coherent processes, methods and tools. In this section, we describe the framework we have developed to manage a PL. Our proposal allows a very broad acceptance of the notion of PL among the multiple actors involved. We have drawn inspiration from a number of existing proposals [6, 13].

4.1 Processes

The product line strategy relies on the following major processes: Build the PL, configure a product in the PL and maintain & enrich the PL.

4.1.1 Build the PL

Companies want to have competitive advantages, and to answer more and more customer needs with individual adaptation. Before designing a system, system engineers have to analyze needs. They examine the system's environment and identify interactions, constraints and available resources. The capture of stakeholder's needs is the key to system engineering. It is also the first source of variants. Thus, in a PL process, the outcome is not only a set of elicited needs, but also a variability assessment of the PL. Needs capture and analysis is combined with the analysis of the PL variability.

Building the PL includes:

- defining a set of assets (components, software, models...) that are designed with the target of addressing a wide range of user needs.
- Building a feature model that describes product features and constraints between them.
- Associating assets to features.

Assets are associated to features with the target of ensuring modularity and enabling evolutivity criteria mentioned in section 2.

4.1.2 Configure a PL

Products to be sold to customers are built as configurations of the product line. The PL contains all assets describing possible products. When a product engineer selects a product for a customer, he defines the features of this specific product. Features are selected in the feature model of the product line. Assets of the specific product are obtained as a consequence of the features. PL assets are filtered according to chosen features to obtain the asset lists of the specific product. Thus, the specific product is a configuration of the product line. This process accounts for the reusability criterion mentioned in section 2.

4.1.3 Maintain & enrich the PL

When engineers have to design a solution for new customer needs or new project requests, they first try to integrate in their design existing assets from the product line. And this shall be done on the system as a whole, considering needs that shall be satisfied, and at the component level for component functionalities and tested qualities. Thus, the design method consists in searching within the existing assets which ones could be reused as they are, which ones could be reused with only small modifications or additional tests and which ones could be integrated in the product via the adaptation of some interfaces or the use of adaptation parts (brackets, connectors, embedding…).

New assets are developed only if existing assets do not allow to answer the new elicited needs. And if so, they are designed in a way to enable their reuse for future products. Each time a new component is developed, it can be included in the asset library. Asset's characteristics are standardized and recorded in the aim of reuse.

The product line also has to be maintained, meaning obsolescence of PL assets is monitored and new assets are developed in the right schedule to replace the obsolete ones without shortage. This process accounts for the evolutivity criterion mentioned in section 2.

4.2 Method

The method we present is intended to describe the product line management and configuration for people who are not necessarily familiar with software development tools. To do so, the steps of the product line use are made visual with graphical diagrams and simple colors codes.

4.2.1 Association between assets and features

Assets have to be associated to features in order to model the transition from the asset library to the selected assets constituting a specific product. For each asset, a logical statement defines its presence. This statement uses logical connectors and features. In that way it is possible to use a feature configuration to filter the asset library and obtain the assets of a specific product. The introduction of the feature conditions in the description of the behavior of components allows for a configurable behavioral model of the product line. The granularity of the features and the association to groups of assets in the library strongly influences the number of operations to be done to configure a product. It has a major impact on the operability criterion introduced in section 2. To define a product, it will not be necessary to select individual parts but product features. If features are well structured, then their choice is operable by humans. As an illustration, a limited set of feature choice is proposed to a customer purchasing a car, impacting the presence of dozens of components in the product.

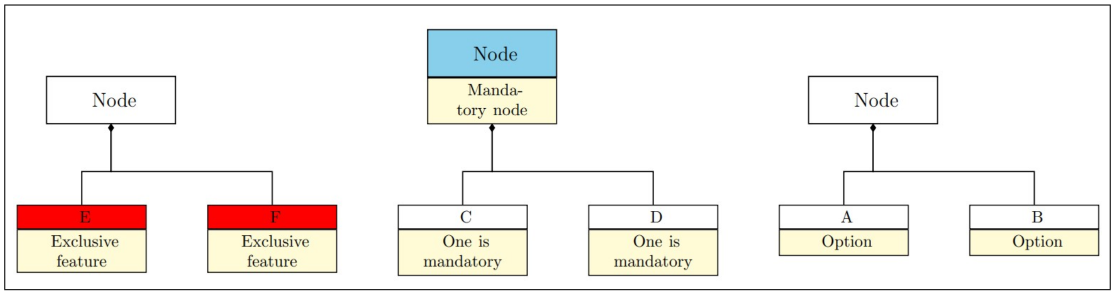


Figure 1: Representation of the different kinds of choice in the feature model

4.2.2 Constraints expressions in a feature diagram

Each node in the feature tree represents a decision. At each decision step, constraints are limiting the number of possible choices. In many cases, the constraints concern neighbor nodes. Thus, being able to display in an intuitive way these constraints is of interest. The color of a node can be used to display these constraints. In our method we propose to consider tree type of choices that are displayed by a color code. The color code can be replaced by any other graphical characteristic of the boxes, especially if accessibility for color-blind users is required.

Optional features are features that can be selected or not without further constraint. Those features are represented in white boxes as shown in Figure 1. A white box can be selected or discarded independently of neighbor boxes. If the parent node is selected, then the children selection can be a single child, both of them or none.

Blue is used as shown in Figure 1 to indicate a mandatory choice: the parent (blue) node has to be kept and at least one of the boxes below has to be selected. When using this colour code, it was found more intuitive for non-specialists to have the blue colour on the upper node, where the decision is taken, rather than on the lower node.

Red is used as shown in Figure 1 to indicate exclusive options: only one of the neighboring red boxes can be selected. If one is selected, then the neighboring red boxes have to be discarded. The red color is applied on lower nodes. It is a difference with the blue color and was found more intuitive for non-specialist users who have to define a product within the PL. Plus, it allows to combine easily red and blue nodes.

4.2.3 Product configuration

The most natural way to fill a variant tree is top down. One begins with the upper node and goes along the branches down to the leaves. At each step, the possible choices are defined by the color of the surrounding boxes. The variant selection process is made visual as displayed in Figure 2 and intuitive to users that have to do it.

A product configuration is obtained by the selection of a set of features that drive the selection of the associated assets. The method to obtain such a configuration requires a set of decisions, whether to keep or not each feature. A feature model is a set of possible decisions, containing also mandatory features. It is important to have them as they can imply sub-decisions. Our feature model defines variable features and constraints between them. In a product configuration, some features are selected, and others are discarded. We use the green color to indicate that a feature is selected, and the gray color to specify that a feature is discarded from a specific product. Thus, a feature model fully colored with green and gray like the one displayed at the bottom of in Figure 2 is a description of an individual product of the product line.

Defining a product configuration and verifying at the end if compatibility rules are satisfied is likely to produce configurations incompatible with the rules. The selection has to be organized in a succession of decisions. To carry out this selection, the feature model is scanned down from the root to the leaves. When a node is discarded, then the whole branch below the node is discarded as well. So there cannot be an alternation of green an gray colors down a branch. Coherency rules are checked at each decision step. If a decision would lead to inconsistent feature selection, then the corresponding choice is not possible. Figure 2 illustrates successive decision steps that lead to a configuration.

Feature selection gives a progressive coloring of the feature model with green and gray. The selection process can be interrupted at any moment. Partially selected feature model can be produced, in which some parts are selected, some others are discarded and some others are still open options. The color of the boxes provides a comprehensive way to describe partial configuration and to define rules to continue the selection process.

4.3 Tool

As the processes to build a new product rely on system engineering, it is natural to use system engineering tools for PL engineering. Thus, variant management shall be embedded in the system engineering tool. One key success factor is to make people working in different domain understand each other and communicate efficiently. Therefore, it is crucial to provide graphical views intended to be intuitive for non-specialists and so to fulfill the simplicity criterion in section 2. We have developed a private model-based system engineering tool that is already in use [18, 7, 8, 21]. In a true digital twin perspective behavioral models can be amended by the feature selection. A model of the feature selection is embedded in the tool. In this perspective, we obtain a model of the product line that describes all the products in the product line with their individual characteristics and behavior. While graphical views presented below are available in the tool, the interpretation of constraints stated as logical formulas is still under development.

5 Discussion

In the former paragraph, we have proposed a description for a feature model. Its semantics is similar to the one proposed by [23]. As different boxes may carry the same label, it describes a Directed Acyclic Graph (DAG). In addition, the colors are used to express “require” and “exclude” relations. Let’s take a closer look at how these constraints are used.

It is clear that the “exclude” relation between neighbor nodes can be directly expressed with the color code. Let A and B be two features in different branches of a variant model. The sentence “Feature A excludes Feature B” means that if feature A is selected, then feature B cannot be selected (let us note that A excludes B is equal to B excludes A). Figure 3 shows how this constraint can be expressed in our colored box language. Beside the main feature tree, we introduce a new branch labeled “constraints” with node A and B within red boxes, meaning the user has to make an exclusive choice. If the user selects the first A node, then through a decision propagation, the other node labeled A is automatically selected because it has the same label. Node B is automatically discarded because of mutual exclusion with A.

The sentence “Feature A requires Feature B” means that if feature A is selected, then feature B

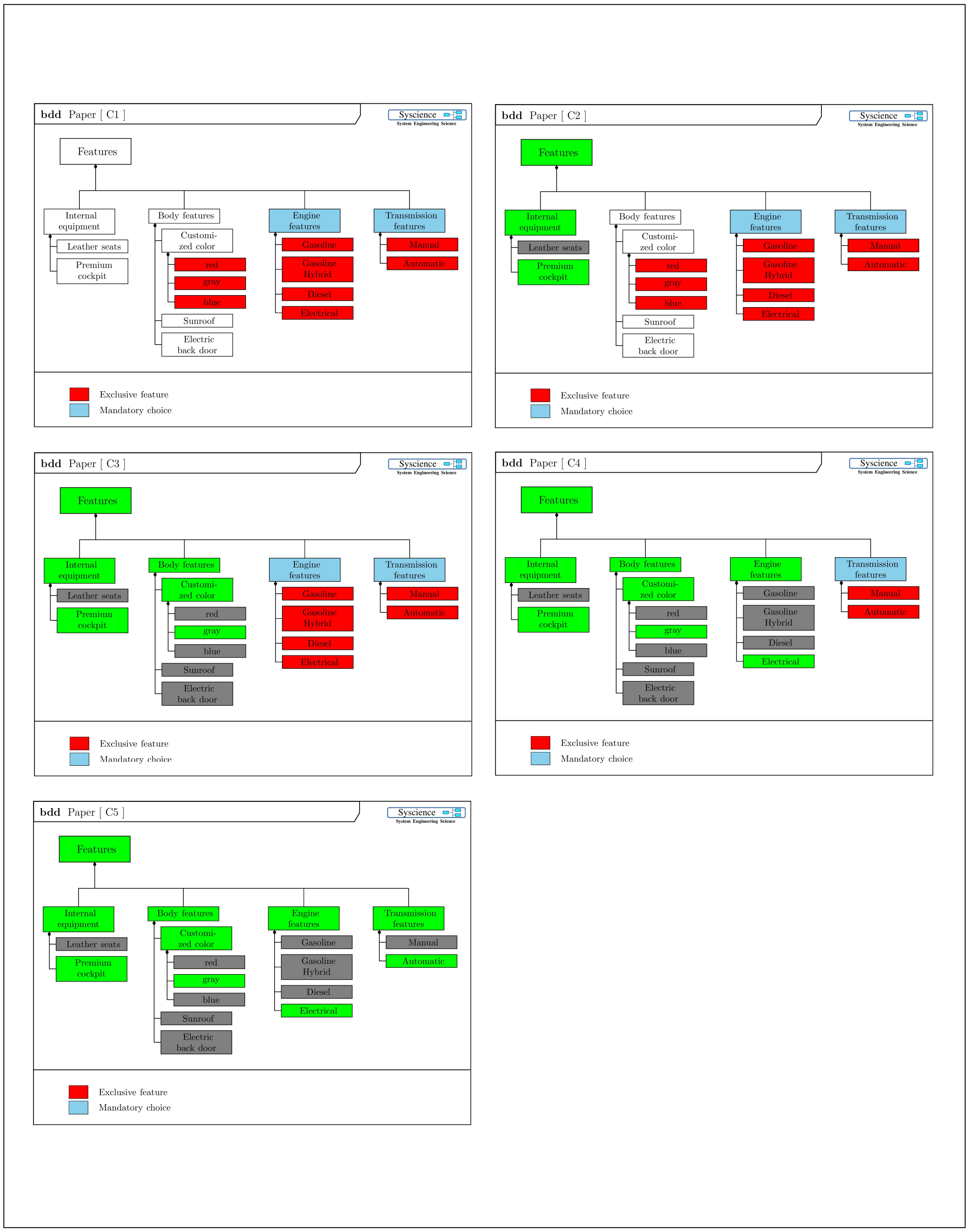


Figure 2: Representation of successive decisions in the feature model

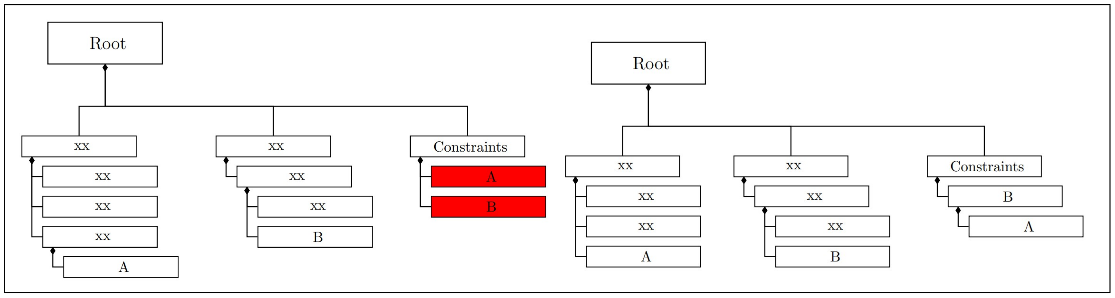


Figure 3: Representation of the constraints A excludes B and A requires B when A and B are in different branches

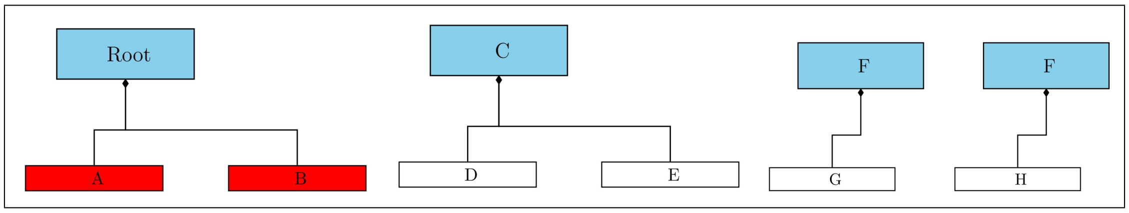


Figure 4: Representation of the logical constraints A is NOT(B), C implies (D OR E) and F implies (G AND H)

is mandatory. Figure 3 shows how this constraint can be expressed in our colored box language. We introduce a new branch labeled "constraints" with a node labeled B. Below this node, a single node labeled A is placed. When the user selects A somewhere in the tree, then decision propagation selects automatically all nodes labeled A. B is automatically selected as a parent of A.

Thus, our graphical language is able to describe the "require" and the "exclude" relations.

Let us express basic logical constraints with this language. Fig. 9 defines a variable B, that corresponds to NOT(A), a variable C that corresponds to (D OR E), and a variable F that corresponds to (G AND H). Since (AND, OR, NOT) is a complete set of connectors in Boolean logic, any Boolean formula can be expressed by this language. For example a nested constraint ((A AND B) => C) has first to be written as a normal disjunctive formula. After that, it is possible to express it with the colour code. Figures 4, gives a general pattern that makes it possible to transform a Boolean expression into a graph with our color definition. The expressiveness of the defined language is universal as soon as we consider feature models that are directed graphs rather than trees.

Circular constraints are not allowed and should be detected by the tool. When the product line is well defined, the selection of an individual product is straightforward. By doing so, the consistency criterion mentioned in section 2 is based on the decision propagation between nodes. If a choice leads to a dead-end where any choice left would violate a constraint, then it is necessary to backtrack and undo the last choice. It can be a future improvement to analyze the structure of constraints and disable any choice leading to a dead-end.

6 Conclusion

In this paper we have discussed the product line approach as a way to design products that can be configured according to customer needs. The product line approach is suited for the automotive industry.

However the large number of actors involved in the definition of product variants and product configuration is limiting the use of complex tools. We have therefore defined a framework for product line management in order to address this issue. The proposed method is based on a color code that makes possible decisions visual and intuitive for users unfamiliar with variant management. The method has been illustrated with an example. Our model-based system engineering tool was used to draw the diagrams. The completeness of the method was discussed.

The current development of our tool includes an allowed configuration only if logical constraints are satisfied at each decision step. In that way, a correct product line feature model does not require a solver to check coherence while configuring a product, as constraints are taken into account at each decision step. The scaling to larger product lines relies on a well structured feature model, broken down into as many sub-trees as necessary to keep each graphical view understandable.

We plan to apply our approach of introducing a product line when defining an initial requirement for a system in an industrial domain, using the platform under development.

Acknowledgment

The authors are grateful to Caroline CABY for her insights and the very fruitful discussions she was involved in.

References

[1] IEEE Std 1220 (2005 (revision of 1998 standard)): *Standard for Application and Management of the Systems Engineering Process*. *IEEE Std 1220*, doi:10.1109/MC.2006.164.

[2] IEEE 1471 (2000): *Recommended Practice for Architectural Description of Software-Intensive Systems*. *IEEE 1471*, doi:10.1109/IEEESTD.2000.91944.

[3] ISO/IEC/IEEE 15288 (2008): *Systems Engineering - System Life Cycle Processes*. *ISO/IEC/IEEE 15288*, doi:10.1109/IEEESTD.2008.4475828.

[4] EIA 632 (2005): *Processes for engineering a system*. *Electronics Industry Association*, doi:10.4271/EIA632.

[5] M. H. ter Beek, K. Schmid & H. Eichelberger (2019): *Textual Variability Modeling Languages: An Overview and Considerations*. In: *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B*, SPLC '19, Association for Computing Machinery, New York, NY, USA, p. 151–157, doi:10.1145/3307630.3342398.

[6] D. Benavides, A. Ruiz-Cortés, P. Trinidad & S. Segura (2006): *A Survey on the Automated Analyses of Feature Models*. In J. C. Riquelme Santos & P. Botella, editors: *XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD, Sitges, Barcelona, Spain*, pp. 367–376. Available at `https://api.semanticscholar.org/CorpusID:17186402`.

[7] S. Berthier & P. Krapf (2020): *Appréhension des risques engendrés par le réchauffement climatique au moyen de l'outil d'Ingénierie Système « L'Atelier Syscience »*. In: *Congrès de maîtrise des risques et de sûreté de fonctionnement, lambda-mu22, 10/11/2020, Le Havre, FRANCE*, pp. 1–10. Available at `https://hal.science/hal-03348084`.

[8] S. Berthier, P. Krapf & C. Oukil (2022): *Analyse des risques induits par le changement climatique sur l'outil productif : Apport d'une approche système*. In: *Congrès Lambda Mu 23 - 23e Congrès de Maîtrise des Risques et de Sûreté de Fonctionne-ment: Innovations et maîtrise des risques pour un avenir durable, IMDR (Institut pour la maîtrise des risques), Oct 2022, Paris Saclay, FRANCE*, pp. 1–10. Available at `https://hal.science/hal-03966604`.

[9] D. Beuche (2007): *Modeling and Building Software Product Lines with pure: : variants*. In: *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings. Second Volume (Workshops)*, Kindai Kagaku Sha Co. Ltd., Tokyo, Japan, pp. 143–144.

[10] C. Caby (2022): *Étude du développement de lignes de produits – Proposition d'une méthode*. Master's thesis, CNAM, Centre régional associé de Bretagne.

[11] P. Clements & L. Northrop (2001): *Software Product Lines: Practices and Patterns*. *Addison-Wesley Professional*. 978-0-201-70332-0.

[12] E. Coşkun, B. Gezici, M. Aydos, A.K. Tarhan & V. Garousi (2022): *ERP failure: A systematic mapping of the literature*. *Data & Knowledge Engineering* 142, doi:10.1016/j.datak.2022.102090.

[13] K. Czarnecki, M. Antkiewicz, C. H. P. Kim, S. Lau & K. Pietroszek (2005): *Model-driven software product lines*. In Ralph E. Johnson & Richard P. Gabriel, editors: *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, ACM, pp. 126–127, doi:10.1145/1094855.1094896.

[14] K. Czarnecki & C. H. P. Kim (2005): *Cardinality-based feature modeling and constraints: A progress report*. In: *International Workshop on Software Factories*, ACM San Diego, California, USA, pp. 16–20. Available at `https://api.semanticscholar.org/CorpusID:12376065`.

[15] Pure systems GmbH (2007): *Variant Management with pure : : variants pure-systems*. Available at `https://api.semanticscholar.org/CorpusID:15988361`.

[16] K. Kang, S. Cohen, J. Hess, W. Nowak & S. Peterson (1990): *Feature-Oriented Domain Analysis (FODA) Feasibility Study (Report)*. *Pittsburgh: Software Engineering Institute, Carnegie Mellon University*. Available at `http://www.sei.cmu.edu/reports/90tr021.pdf`.

[17] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz & S. Apel (2009): *FeatureIDE: A tool framework for feature-oriented software development*. In: *2009 IEEE 31st International Conference on Software Engineering*, pp. 611–614, doi:10.1109/ICSE.2009.5070568.

[18] P. Krapf, S. Rakotosolofo & S. Berthier (2018): *Utilisation d'un atelier d'ingénierie système pour l'Identification des risques d'un véhicule connecté*. In: *Congrès de maîtrise des risques et de sûreté de fonctionnement, lambdamu21, 16-18/10/2018, Reims, FRANCE*, pp. 1–10. Available at `https://hal.archives-ouvertes.fr/hal-02073215`.

[19] A.M. Madni, C.C. Madni & S.D. Lucero (2019): *Leveraging Digital Twin Technology in Model-Based Systems Engineering*. *Systems* 7(1), doi:10.3390/systems7010007.

[20] NASA (1995): *Systems Engineering Handbook*. *NASA*. SP-610S.

[21] C. Oukil, P. Krapf & S. Berthier (2022): *Analyse et évaluation des risques liés à la mise à jour des logiciels de la voiture autonome*. In: *Congrès Lambda Mu 23 « Innovations et maîtrise des risques pour un avenir durable » - 23e Congrès de Maîtrise des Risques et de Sûreté de Fonctionnement, Institut pour la Maîtrise des Risques, Oct 2022, Paris Saclay, FRANCE*, pp. 1–10. Available at `https://hal.science/hal-03877941v1`.

[22] A. Le Put (2014): *L'ingénierie système d'une ligne de produits*. *Cépadues Editions*. ISBN : 9782364931220.

[23] P-Y Schobbens, P. Heymans, J-C Trigaux & Y. Bontemps (2007): *Generic Semantics of Feature Diagrams*. *Computer Networks* 51(2), doi:10.1016/j.comnet.2006.08.008.

[24] L. Wozniak & P. Clements (2015): *How automotive engineering is taking product line engineering to the extreme*. In Douglas C. Schmidt, editor: *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, ACM, pp. 327–336, doi:10.1145/2791060.2791071.