Simulation-Based Abstractions for Software Product-Line Model Checking

Cordy, Maxime; Classen, Andreas; Perrouin, Gilles; Heymans, Patrick; Schobbens, Pierre-Yves; Legay, Axel

Publication date:
2011

Document Version
Early version, also known as pre-print

Simulation-Based Abstractions for Software Product-Line Model Checking

Maxime Cordy[(1) *] Andreas Classen[(1) †] Gilles Perrouin[(1)]
Patrick Heymans[(1)(2)] Pierre-Yves Schobbens[(1)]
Axel Legay[(3)(4)(5)]

[(1)] University of Namur, Belgium
[(2)] INRIA Lille-Nord Europe, Université Lille 1 – LIFL – CNRS , France
[(3)] INRIA Rennes, France
[(4)] Aalborg University, Denmark
[(5)] University of Liège, Belgium

Abstract

Software Product Line (SPL) engineering is a software engineering paradigm that exploits the commonality between similar software products to reduce life cycle costs and time-to-market. Many SPLs are critical and would benefit from efficient verification through model checking. Model checking SPLs is more difficult than for single systems, since the number of different products is potentially huge. In previous work, we introduced Featured Transition Systems (FTS), a formal, compact representation of SPL behaviour, and provided efficient algorithms to verify FTS. Yet, we still face the state explosion problem, like any model checking-based verification. Model abstraction is the most relevant answer to state explosion. In this paper, we define a novel simulation relation for FTS and provide an algorithm to compute it. We extend well-known simulation preservation properties to FTS and thus lay the theoretical foundations for abstraction-based model checking of SPLs. We evaluate our approach by comparing the cost of FTS-based simulation and abstraction with respect to product-by-product methods. Our results show that FTS are a solid foundation for simulation-based model checking of SPL.

Keywords – Model Checking, Software Product Lines, Formal methods, Simulation, Abstraction, Feature.

*FNRS research fellow
†FNRS research fellow

1 Introduction

Software Product Line (SPL) engineering is an increasingly popular software development paradigm targeting families of similar software products. It allows to make substantial economies of scale by taking into account the commonalities between the family members during the whole development life cycle. The different variants of the system (called products) are identified upfront and a model of their differences and commonalities – typically a feature diagram [1] – is created. In this context, features are atomic units of difference that appear natural to stakeholders and technicians alike. SPL engineering has become widespread in industry, including critical applications such as automotive or avionics. These software products require solid evidence that they work correctly according to their requirements and intended properties.

Model checking [2] is a well-known technique for verifying system behaviour. A simple method for checking a product line consists in applying single-system model checking algorithms [3, 4] to each individual product. However, for an SPL with n features, up to 2^n executions of those algorithms may be needed. This *enumerative* approach is clearly impractical and thus should be replaced by new verification approaches specific to product lines.

In our previous work [5–7], we addressed this problem by introducing *Featured Transition Systems* (FTS) – see example in Figure 1. FTS are an extension of transition systems that represent the behaviour of *all* the products of a given SPL in a compact structure. We also proposed FTS-specific model checking algorithms to verify the whole SPL in a single execution. More precisely, these algorithms model check the SPL against temporal properties expressed either in *Linear Time Logic* (LTL) [8] extended with features (fLTL) or in an extended *Computation Tree Logic* (CTL) [9], fCTL. These logics can be used to express properties such as: *for all products with features f and g, a request α is always followed by a response β.* Given such a property, our algorithms can compute the features required for the property to be satisfied, and hence the products of the SPL that do satisfy the property. We call them *FTS algorithms.*

We evaluated the efficiency of the FTS algorithms through the implementation of several libraries and tools. First, we developed a Haskell library for checking an FTS against LTL formulae [5]. We also built an extension to the model-checker NuSMV to verify CTL formulae using the FTS algorithms [6]. Recently, we developed SNIP [7], an SPL model-checking toolset that combines the FTS algorithms with Promela, the high-level specification language used in the well-known model-checker SPIN [10].

Early experiments have shown that the FTS model-checking approach is more efficient than the enumerative approach. Indeed, when comparing the two approaches implemented within our tools, we observe that FTS algorithms generally outperform the enumerative method [5]. However, our

experiments also have shown that there is still much room for improvement. SPL verification theory is still at an early stage and needs to be further improved to target industry-scale SPL verification.

Model abstraction is an optimisation that aims to simplify a model prior to its verification [11]. Roughly speaking, an abstraction function is used to reduce the size of a model by merging similar states. Depending on the definition of this function, the behaviour of the model may change, that is, new behaviours may appear, existing ones may disappear, or both. Characterising the abstracted model with respect to the original one is therefore essential since those behavioural modifications may impact on the satisfiability of temporal properties. Such a characterisation is generally obtained thanks to the definition of a simulation relation [12].

In this paper, we lay the theoretical foundations for abstraction-based model checking of SPLs. First, we extend the definition of simulation from transition systems to FTS and propose an algorithm that computes this relation. We then establish which properties are preserved by the simulation relation, and for which products. This is required to perform reliable checking. Then, we define three abstractions based on the notion of simulation quotient [13] that can be applied to remove redundant behaviour in an FTS and thus reduce verification time. In addition to abstraction, simulation relations have numerous applications. In particular, simulation-based model checking is an established verification method, as LTL/CTL model checking is. Our solution allows easier verification of properties modelled visually (as automata) rather than logical formulae, which is more suitable for engineers. Studying FTS simulation is thus as important as generalising LTL/CTL model checking to FTS. We provide a concrete implementation for computing simulation and applying abstraction to FTS. We carry out a complexity evaluation and empirical evaluations that reveal substantial efficiency improvements over enumerative application of classical simulation. This corroborates previous results by characterising the gain of FTS-based simulation model checking over enumerative, TS-based, simulation model checking.

The structure of the paper is as follows. In Section 2, we recall essential results, theorems and properties related to the abstraction of transition systems, as well as the definition of FTS. Section 3 is focused on the definition and the computation of the simulation relation. Section 4 defines the simulation quotient as well as abstraction functions based on it. Section 5 describes or experiments and their results. Finally, Section 6 presents related work in the fields of SPL modelling and abstraction-based verification.

2 Background

In this section, we first present the established concepts related to the verification and the abstraction of *Transition Systems* (TS). TS are a classical behavioural model for single systems. We also briefly recall some definitions of our previous work [5, 6] that are needed in the paper.

2.1 Single-Product Model Checking

Model checking is a well-known technique for verifying both hardware and software against temporal properties. Basically, given the model of a system M and a temporal property Φ, a model-checking algorithm determines whether or not M satisfies Φ, written $M \models \Phi$. For single systems TS are used and are defined as follows [13].

Definition 1 *A TS is a tuple* $(S, trans, I, AP, L)$ *where* S *is a set of states,* $trans \subseteq S \times S$ *is a transition relation,* $I \subset S$ *is a set of initial states,* AP *is a set of atomic propositions, and* $L : S \to 2^{AP}$ *is a labelling function that associates every states with the set of atomic propositions satisfied by this state.*

We call a behaviour of the system the sequence of atomic propositions satisfied during its execution. The semantics of a TS, noted $[\![ts]\!]_{TS}$, is then its set of behaviours, that is

$$[\![ts]\!]_{TS} = \{L(s_0), L(s_1), \dots \mid s_0 \in I \wedge (s_i, s_{i+1}) \in trans\}.$$

Note that the definition of TS usually includes a set of actions. However, these are not considered in this paper and consequently, they are ignored in our definition, which thus boils down to a Kripke structure.

TS can model a software product at different abstraction levels. If a more abstract (that is, smaller) model preserves the properties of a larger model, it is more efficient to check properties on the abstract model. It is therefore essential to be able to relate two models at different abstraction levels. For single systems, this information is formally captured by a *simulation relation* [12].

Definition 2 *Let* $TS_i = (S_i, trans_i, I_i, AP, L_i), i \in \{1, 2\}$ *be transition systems over* AP*. A simulation for* (TS_1, TS_2) *is a binary relation* $\mathcal{R} \subseteq S_1 \times S_2$ *such that*

1. $\forall s_1 \in I_1 \bullet \exists s_2 \in I_2 \bullet (s_1, s_2) \in \mathcal{R}$ *and*
2. $\forall (s_1, s_2) \in \mathcal{R}$ *it holds that*
   (a) $L_1(s_1) = L_2(s_2)$ *and*
   (b) $\forall s_1' \in Post(s_1) \bullet \exists s_2' \in Post(s_2) \bullet (s_1', s_2') \in \mathcal{R}$.

where $Post(s) = \{s_2 | trans(s, s_2)\}$ *denotes the set of states that can be reached from* s*. Then,* TS_2 *simulates* TS_1*, denoted by* $TS_1 \preccurlyeq_{TS} TS_2$ *if there exists a simulation for* (TS_1, TS_2)*.*

According to this definition, if TS_1 is simulated by TS_2, then any behaviour of TS_1 can be reproduced in TS_2. We can extend this definition to couples of states instead of couples of TSs. In this case, state s_1 is simulated by state s_2 iff $(s_1, s_2) \in \mathcal{R}$ for some $\mathcal{R}$, also noted $s_1 \preccurlyeq_{TS} s_2$. Intuitively, this means that any behaviour produced from s_1 can be produced from s_2. Also, $\preccurlyeq_{TS}$ is a preorder – it is reflexive and transitive [13]. Additionaly, when $TS_1 \preccurlyeq_{TS} TS_2$ and $TS_2 \preccurlyeq_{TS} TS_1$, the two transition systems are called *simulation-equivalent*, noted $TS_1 \simeq_{TS} TS_2$. Intuitively, this means that TS_1 and TS_2 model exactly the same behaviour. Since $\preccurlyeq_{TS}$ is a preorder, $\simeq_{TS}$ is an equivalence relation [13].

The definition of simulation allows one to characterise the behaviour of an abstract transition system $\widehat{ts}$ with regard to an original model ts. Informally, an abstract transition system is obtained by merging states for which a so-called abstraction function returns the same value. The abstraction may add or remove behavioural options, depending on the chosen abstraction function. However, a relevant analysis requires to have either $ts \preccurlyeq_{TS} \widehat{ts}$, $ts \preccurlyeq_{TS} \widehat{ts}$ or both. If this condition is satisfied, we can show that the abstraction preserves the (un)satisfiability of properties of a certain type.

Indeed, there is a strong link between the simulation relation and the properties satisfied by two TSs. In this paper, we focus on properties expressed in *Linear Time Logic* (LTL) [8]. However, the presented results can also be applied to specific fragments of the *Computation Tree Logic* (CTL) [9]. If a simulation relation exists between two transition systems, we can show that an LTL formula satisfied by the simulating TS is preserved in the simulated one [12, 13].

Property 3 *Let* TS_1 *and* TS_2 *be two transition systems without terminal states and* Φ *an LTL property. Then,*

$$TS_1 \preccurlyeq_{TS} TS_2 \Rightarrow (TS_2 \models \Phi \Rightarrow TS_1 \models \Phi).$$

The following statement is equivalent: if TS_1 does not satify Φ, neither does TS_2. Finally, if $TS_1 \simeq_{TS} TS_2$, then they satisfy exactly the same LTL properties. In particular, if TS_2 is an abstraction of TS_1, proving that the abstract TS verifies an LTL formula suffices to ensure that the formula holds for TS_1. Therefore, abstraction can drastically shorten the time and space cost of verification.

2.2 Software Product Line Verification

While a TS is convenient to model the behaviour of an individual product of an SPL, it is not suitable for concisely representing all the possible products. To overcome this, we defined *Featured Transition Systems* (FTS) [5].

Basically, an FTS is a TS augmented with transitions labelled with feature expressions (see Figure 1). These features are described in a *feature diagram* (FD) that establishes the set of legal products [1, 14]. For this paper, it is enough to know that the semantics of a feature diagram d defined over a set of features N is the set of all the valid products, that is a set of sets of features, denoted by $[\![d]\!]_{FD} \subseteq \mathcal{P}(N)$. Schobbens *et al.* [1] give a more thorough and formal definition of FD.

Also, every transition of an FTS is labelled with a feature expression that defines the products able to execute the transition. Formally, FTS are defined as follows [5, 6].

Definition 4 *An FTS is a tuple* $(S, trans, I, AP, L, d, \gamma)$*, where*

- $S, trans, I, AP, L$ *are defined as in Definition 1,*
- d *is a feature diagram,*
- $\gamma : trans \to \left(\{0,1\}^{|N|} \to \{0,1\}\right)$ *is a total function, labelling each transition with a* feature expression $\in \mathbb{B}(N)$*, i.e., a Boolean function over the set of features. By* $[\![\gamma(t)]\!]$*, we denote the set of products that satisfy* $\gamma(t)$*.*

Similarly with TS, our definition of FTS does not include a set of actions.

An FTS can be seen as the merging of all the TSs of the products that compose the SPL. Because of that, the successor operator must be redefined in order to take into account that a state can be a successor of another one only for a specific set of products [5].

Definition 5 *The successors of* $s \in S$ *for products* $px \subseteq \mathcal{P}(N)$ *are given by*

$$Post(s, px) = \{(s', px') \mid (s, s') \in trans \wedge px' = px \cap [\![\gamma(s, s')]\!]\}.$$

Furthermore, any TS corresponding to a specific product can be obtained from the FTS by applying a *projection* function. In simple terms, this function removes all the transitions of the FTS whose feature expression is not satisfied by the considered product [5].

Definition 6 *The projection of an FTS* fts *to a product* $p \in [\![d]\!]_{FD}$*, noted* $fts_{|p}$*, is the TS* $ts = (S, trans', I, AP, L)$ *where* $trans' = \{t \in trans \mid p \in [\![\gamma(t)]\!]\}$*.*

Because the FTS represents the behaviour of *all* the products, its semantics is defined as a function that associates a product with the set of behaviours of its projection.

Definition 7 *The semantics of an FTS fts is a function $[\![fts]\!]_{FTS}$ with domain $[\![d]\!]_{FD}$ such that*

$$\forall p \in [\![d]\!]_{FD} \bullet [\![fts]\!]_{FTS}(p) = [\![fts_{|p}]\!]_{TS}.$$

Finally, in [6], we extended CTL to define a property only on a subset of the valid products. The same extension can easily be applied to LTL.

Definition 8 *An fLTL property Ψ is an expression $\Psi = [\chi]\Phi$ where $\chi : \{0,1\}^{|N|} \to \{0,1\}$ is a feature expression and Φ an LTL property. An FTS fts satisfies an fLTL property $[\chi]\Phi$ iff*

$$\forall p \in [\![d]\!]_{FD} \cap [\![\chi]\!] \bullet fts_{|p} \models \Phi.$$

3 Simulation relation for SPL models

Since a model abstraction does not necessarily verify the same properties as the original model, it is essential to characterise the behavioural inconsistencies between the model and its abstraction. In the previous section, we presented the simulation for TSs as a computable relation that can establish this characterisation. However, their definition is clearly unsuitable in our context because we are interested in abstracting SPL models rather than models of individual systems. In this section, we thus introduce a definition of simulation for FTS. We also propose an algorithm to compute it and we establish a link between the latter and the preservation of fLTL formulae.

3.1 Simulation Relation for FTS

As a first step, we extend the definition of simulation to FTS. For this purpose, we first impose the restriction that a simulation relation can only hold between two FTS defined *over the same FD*. Intuitively, an FTS simulates another one iff every valid product has more behaviour in the former FTS than in the latter. Formally, this condition can be expressed using the simulation on TS as defined previously.

Definition 9 *Let $fts_i = (S_i, trans_i, I_i, AP_i, L_i, d, \gamma_i)$, $i \in \{1,2\}$, be FTS with $AP_1 \subseteq AP_2$. Then, fts_1 is simulated by fts_2 for products $[\![fts_1 \preccurlyeq_{FTS} fts_2]\!] \subseteq [\![d]\!]_{FD}$, where*

$$[\![fts_1 \preccurlyeq_{FTS} fts_2]\!] = \{p \in [\![d]\!]_{FD} : fts_{1\,|p} \preccurlyeq_{TS} fts_{2\,|p}\}.$$

Since the semantics $(fts_1 \preccurlyeq_{FTS} fts_2)$ is a set of products, we see it as a feature expression. Furthermore, fts_1 is completely simulated by fts_2 iff $[\![fts_1 \preccurlyeq_{FTS} fts_2]\!] = [\![d]\!]_{FD}$.

Note that this definition does not consider illegal products, that is products that are not included in $[\![d]\!]_{FD}$.

Thanks to the above definition, we can already determine for which products an FTS simulates another. However, this would require computing the simulation relation on TS for $\mathcal{O}(2^n)$ couples of TS, which sums up to an overall time complexity bounded by $\mathcal{O}(|S|^4.2^n)$ [13]. Instead, we aim to take advantage of the compact structure of FTS, as we did for solving the model checking problem for SPL in our previous work [5, 6]. Hence, we propose the following alternative definition.

Definition 10 *Let $fts_i = (S_i,\ trans_i,\ I_i,\ AP_i,\ L_i,\ d,\ \gamma_i)$, $i \in \{1, 2\}$, be featured transition systems with $AP_1 \subseteq AP_2$. A simulation for (fts_1, fts_2) is a binary function $\mathcal{R} : S_1 \times S_2 \to \mathbb{B}(N)$ such that*

$$\mathcal{R}(s_1, s_2) = (L_1(s_1) = (L_2(s_2) \cap AP_1)) \wedge \bigwedge_{s_1'} \mathcal{R}_{via}(s_1 \to s_1', s_2)$$

where $\mathcal{R}_{via}(s_1 \to s_1', s_2)$ is given by

$$\gamma_1(s_1, s_1') \Rightarrow \bigvee_{s_2'} (\mathcal{R}(s_1', s_2') \wedge \gamma_2(s_2, s_2'))$$

with $(s_1, s_1') \in trans_1$ and $(s_2, s_2') \in trans_2$.

Then, fts_1 is simulated by fts_2 for products $[\![\bigwedge_{i_1 \in I_1} \bigvee_{i_2 \in I_2} \mathcal{R}_{FTS}(i_1, i_2)]\!] \subseteq [\![d]\!]_{FD}$ where $\mathcal{R}_{FTS}$ is the largest simulation for (fts_1, fts_2). By largest, we mean that for any states s_1, s_2 and simulation $\mathcal{R}$ for (fts_1, fts_2), we have $[\![\mathcal{R}(s_1, s_2)]\!] \subseteq [\![\mathcal{R}_{FTS}(s_1, s_2)]\!]$.

This definition can be seen as a generalisation of Definition 2. Intuitively, $[\![\mathcal{R}_{FTS}(s_1, s_2)]\!]$ contains only products for which s_2 has more behaviour than s_1. In other words, for each product $p \in [\![\mathcal{R}_{FTS}(s_1, s_2)]\!]$ and transition (s_1, s_1') available for p, s_2 must have at least one successor s_2' reachable by p and such that $p \in [\![\mathcal{R}_{FTS}(s_1', s_2')]\!]$. Similarly, $[\![\mathcal{R}_{via}(s_1 \to s_1', s_2)]\!]$ contains only products for which s_2 can simulate the transitions from s_1 to s_1'. Let us note that, according to our definition:

- $[\![\mathcal{R}_{FTS}(s_1, s_2)]\!] \neq \emptyset$ implies that all atomic proposition satisfied by s_1 (that is, propositions in AP_1) are satisfied by s_2.
- for given s_1 and s_1' such that $\mathcal{R}_{via}(s_1, s_2, s_1') = (px \vee px')$, it may happen that s_2 simulates a transition $s_1 \to s_1'$ for products px via a transition $s_2 \to s_2'$ and for products px' thanks to another transition $s_2 \to s_2''$.

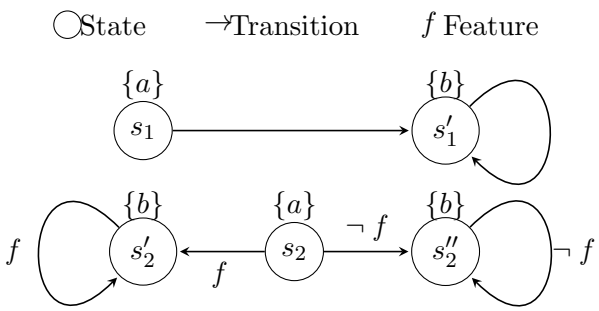

Figure 1: Two simulation-equivalent FTS.

To illustrate this second remark, let us consider the basic example presented in Figure 1. We observe that $\mathcal{R}_{FTS}(s'_1, s'_2) = f$ and $\mathcal{R}_{FTS}(s'_1, s''_2) = \neg f$. Furthermore, s_2 simulates s_1 for products $[\![f]\!]$ through transition (s_2, s'_2) and for products $[\![\neg f]\!]$ via transition (s_2, s''_2). In the end, we conclude that s_2 simulates s_1 for all products.

As established in the following theorem, the two definitions of simulation for FTS we have given are equivalent.

Theorem 11 *Let fts_i, $i \in \{1, 2\}$, be FTS. Let $\mathcal{R}_{FTS}$ be the largest simulation for (fts_1, fts_2). Then, it holds that*

$$[\![fts_1 \preccurlyeq_{FTS} fts_2]\!] = [\![\mathcal{R}_{FTS}(fts_1, fts_2)]\!].$$

Proof. *First, let us remark that $\forall p \in (fts_1 \preccurlyeq_{FTS} fts_2) \bullet fts_{1\,|p} \preccurlyeq_{TS} fts_{2\,|p}$. Thus, for each product p, according to Definition 2, there exists a relation $\mathcal{R}_{TS}$ such that $\forall i_1 \in I_1 \bullet \exists i_2 \in I_2 \bullet \mathcal{R}_{TS}(i_1, i_2)$ where for all $s_1, s_2 \in S$, $\mathcal{R}_{TS}(s_1, s_2)$ is given by*

$$(L(s_1) = L(s_2) \cap AP_1) \wedge \\ \forall s'_1 \in Post(s_1, \{p\}) \bullet \exists s'_2 \in Post(s_1, \{p\}) \bullet \mathcal{R}_{TS}(s'_1, s'_2).$$

Let us suppose that for all p, $\mathcal{R}_{TS}$ is the largest relation satisfying this equation.

According to Definition 10, we have that $p \in [\![\bigwedge_{i_1 \in I_1} \bigvee_{i_2 \in I_2} \mathcal{R}_{FTS}(i_1, i_2)]\!]$ iff $\forall i_1 \in I_1 \bullet \exists i_2 \in I_2 \bullet p \in [\![\mathcal{R}_{FTS}(i_1, i_2)]\!]$. For a product $p \in [\![\mathcal{R}_{FTS}(s_1, s_2)]\!]$, any transition (s_1, s'_1) available to p, there is a transition (s_2, s'_2) that p can execute such that s'_2 simulates s'_1 for p, that is $p \in \mathcal{R}_{FTS}(s'_1, s'_2)$. Hence, $p \in [\![\mathcal{R}_{FTS}(s_1, s_2)]\!]$ is equivalent to

$$(L(s_1) = L(s_2) \cap AP_1) \wedge \\ \forall s'_1 \in Post(s_1, \{p\}) \bullet \exists s'_2 \in Post(s_1, \{p\}) \bullet p \in [\![\mathcal{R}_{FTS}(s'_1, s'_2)]\!].$$

Thus, $p \in [\![\mathcal{R}_{FTS}(s_1, s_2)]\!]$ implies that there exists a relation $\mathcal{R}_{TS}$ for p. Furthermore, since $\mathcal{R}_{FTS}$ is the largest FTS-simulation for (fts_1, fts_2), we also have that $\mathcal{R}_{TS}$ exists for p implies that $p \in [\![\mathcal{R}_{FTS}(s_1, s_2)]\!]$.

As for TS, we can define simulation-equivalence for FTS. Intuitively, two FTS are simulation-equivalent for products px iff they have the same behaviour for all these products.

Definition 12 *Let fts_1 and fts_2 be two FTSs. If $[\![fts_1 \preccurlyeq_{FTS} fts_2]\!] = px$ and $[\![fts_2 \preccurlyeq_{FTS} fts_1]\!] = px'$ then fts_1 and fts_2 are called simulation-equivalent for products in $[\![fts_1 \simeq_{FTS} fts_2]\!] = px \cap px'$. Furthermore, they are called completely simulation-equivalent iff $[\![fts_1 \simeq_{FTS} fts_2]\!] = [\![d]\!]$.*

3.2 Computing the Simulation Relation

We proposed two equivalent definitions of the simulation relation for FTS. While the former is more intuitive, it is cumbersome to compute for a large SPL, *i.e.* with a high number of products. On the contrary, the latter takes advantage of the compact structure of the FTS. In this subsection, we present a method to compute the relation using this second definition. Basically, $\mathcal{R}_{FTS}$ is obtained, for all couples of states of a given FTS, by computing the greatest fixed point of a function.

First, we need to define a partial order $\leq$ on the feature expressions. Let e and e' be two feature expressions. We say that e is included in e', noted $e \leq e'$, iff $[\![e]\!] \subseteq [\![e']\!]$. Using this partial order, we define that $\mathcal{R}$ is included in $\mathcal{R}'$, noted $\mathcal{R} \subseteq \mathcal{R}'$, iff

$$\forall (s_1, s_2) \bullet \mathcal{R}(s_1, s_2) \leq \mathcal{R}'(s_1, s_2).$$

Then, the simulation function can be computed as the greatest fixed point of the equations of $\mathcal{R}$ in Definition 10, denoted by $T(\mathcal{R})$. Note that $\forall R \bullet \forall i \geq 0 \bullet T(\mathcal{R}) \subseteq \mathcal{R}$. Then according to the Knaster–Tarski theorem, $\mathcal{R}_{FTS}$ can be computed as follows:

$$\mathcal{R}_{FTS} = \mathcal{R}_i \bullet \forall j \geq i \bullet \mathcal{R}_i = \mathcal{R}_j$$

with $\forall s_1, s_2$

$$\mathcal{R}_0(s_1, s_2) = \begin{cases} \mathbb{B}([\![d]\!]_{FD}), & L(s_1) = L(s_2) \cap AP_1 \\ \mathbb{B}(\emptyset), & \text{otherwise} \end{cases}$$

$$\mathcal{R}_{i+1}(s_1, s_2) = T(\mathcal{R}_i)(s_1, s_2)$$

where for a set of products px, $\mathbb{B}(px)$ denotes a feature expression such that $[\![\mathbb{B}(px)]\!] = px$.

Thanks to this algorithm, we can compute $(fts_1 \preccurlyeq_{FTS} fts_2)$ for any $fts_i = (S_i, Act_i, trans_i, I_i, AP_i, L_i, d, \gamma_i)$. For this purpose, we apply it to the FTS $fts_1 \oplus fts_2$, which is defined by

$$fts_1 \oplus fts_2 = (S_1 \uplus S_2, trans_1 \cup trans_2, \\ I_1 \cup I_2, AP_1 \cup AP_2, L, d, \gamma)$$

where $\uplus$ denotes the disjoint union, $L(s) = L_i(s)$ iff $s \in S_i$, and $\gamma(t) = \gamma_i(t)$ iff $t \in trans_i$. Using the value of $\mathcal{R}_{FTS}$ between each initial state of fts_1 and each initial state of fts_2, we determine $(fts_1 \preccurlyeq_{FTS} fts_2)$. Note that Baier and Katoen [13] present a similar method for computing the simulation relation between two TS.

3.3 Property Preservation

The simulation relation for TS is particularly well-known for its interesting preservation properties [2, 12, 13]. In particular, if TS_2 simulates TS_1, then any LTL property satisfied by TS_2 is also satisfied by TS_1. As we show in this section, a similar results holds for FTS simulation.

First, we must define a new notion of satisfiability specific to product lines. Indeed, the model-checking problem for SPL does more than determining the satisfiability of a formula: it requires to identify all the products that do not satisfy the formula, hence the need of a new satisfiability relation.

Definition 13 *Let fts be a FTS and $\Psi = [\chi]\Phi$ an fLTL property. Then, the F-satisfiability of Ψ by fts, noted $fts \models_{FTS} \Psi$, is a feature expression such that*

$$[\![fts \models_{FTS} \Psi]\!] = \{p \in [\![d]\!]_{FD} : p \in [\![\chi]\!] \Rightarrow fts_{|p} \models \Phi\}.$$

Similarly, we define the F-unsatisfiability of Ψ by fts, noted $fts \not\models_{FTS} \Psi$, as a feature expression such that

$$[\![(fts \not\models_{FTS} \Psi)]\!] = \{p \in [\![d]\!]_{FD} \cap [\![\chi]\!] : fts_{|p} \not\models \Phi\}.$$

It has to be noted that $\{[\![fts \models_{FTS} \Psi]\!], [\![fts \not\models_{FTS} \Psi]\!]\}$ is a partition of $[\![d]\!]_{FD}$. Thanks to Definitions 9, 10, and 13, Property 3 can be generalised to FTS, as established in the following theorem. Again, we omit the proof and refer the reader to our technical report [15].

Theorem 14 *Let $fts_i = (S_i, Act_i, trans_i, I_i, AP, L_i, d, \gamma_i)$, $i \in \{1, 2\}$, be two FTS, $\Psi = [\chi]\Phi$ an fLTL property, and $px = [\![fts_1 \preccurlyeq_{FTS} fts_2]\!]$. Then, it holds that*

$$p \in px \Rightarrow (p \in [\![fts_1 \not\models_{FTS} \Psi]\!] \Rightarrow p \in [\![fts_2 \not\models_{FTS} \Psi]\!])$$
$$p \in px \Rightarrow (p \in [\![fts_2 \models_{FTS} \Psi]\!] \Rightarrow p \in [\![fts_1 \models_{FTS} \Psi]\!]).$$

Proof. *Let us assume that* $p \in [\![(fts_1 \not\models_{FTS} \Psi)]\!] \cap [\![fts_1 \preccurlyeq_{FTS} fts_2]\!]$. *By Definition 9, we have that*

$$\forall p' \in [\![px]\!] : fts_{1\,|p'} \preccurlyeq_{TS} fts_{2\,|p'}.$$

Because of Property 3, we know that

$$fts_{1\,|p'} \not\models \Phi \Rightarrow fts_{2\,|p'} \not\models \Phi.$$

Finally, by Definition 13, we obtain

$$p \in [\![(fts_1 \not\models_{FTS} \Psi)]\!] \Rightarrow p \in [\![(fts_2 \not\models_{FTS} \Psi)]\!].$$

The second property is proven similarly.

In particular, this theorem implies that two completely simulation-equivalent FTS have the same FTS-(un)satisfiability.

4 FTS simulation quotient

In the previous section, we defined a simulation relation for FTS and we established the link between this relation and the F-satisfiability of an fLTL property. Our objective is to present, through the study of simulation quotient, how we can define FTS abstractions. We also make use of the preservation properties (see Theorem 14) to determine how the verification of an abstract FTS provides information about the original system.

4.1 Simulation Quotient

The first abstraction we introduce does not modify the behaviour of the FTS to which it is applied. It merely consists in defining the simulation quotient [13] for FTS. This form of abstraction merges states that are completely simulation-equivalent, *i.e.* for all products in $[\![d]\!]_{FD}$. Formally, we define the binary relation

$$\simeq^d_{FTS} \subseteq S \times S \bullet s_1 \simeq^d_{FTS} s_2 \Leftrightarrow \mathcal{R}_{FTS}(s_1, s_2) = [\![d]\!]_{FD}.$$

This relation is an equivalence, since $\simeq_{TS}$ is also an equivalence relation [13]. Therefore, the state space of any FTS can be partitioned into equivalence classes under $\simeq^d_{FTS}$. Our first abstraction function merges states of the same equivalence class. $[s]_{/\simeq^d_{FTS}}$ denotes the equivalence class of s.

The function associates an FTS fts with an abstracted FTS $fts_{/\simeq^d_{FTS}} = (S', trans', I', AP, L', d, \gamma')$ such that

$$\begin{aligned} S' &= \{[s]_{/\simeq^d_{FTS}}\} \\ trans' &= \{([s]_{/\simeq^d_{FTS}}, [s']_{/\simeq^d_{FTS}}) \\ &\qquad \mid (s, s') \in trans\} \\ I' &= \{[s]_{/\simeq^d_{FTS}} \mid s \in I\} \\ L'([s]_{/\simeq^d_{FTS}}) &= L(s), \\ \gamma'(([s]_{/\simeq^d_{FTS}}, [s']_{/\simeq^d_{FTS}})) &= \bigvee_{s'', s'''} \gamma(s'', s''') \end{aligned}$$

where $s'' \in [s]_{/\simeq^d_{FTS}}, s''' \in [s']_{/\simeq^d_{FTS}}$, and $(s'', s''') \in trans$. It thus requires to compute first the simulation relation for every pair of states in the FTS (see Section 3.2).

Such an abstracted FTS has exactly the same behaviour as the FTS on which the function is applied. Indeed, we merge only states that are simulation-equivalent for every products in $[\![d]\!]_{FD}$. Thus, the merging neither adds nor removes any behaviour. A formal proof is given below.

Theorem 15 *Let fts be an FTS. Then, we have $(fts \simeq_{FTS} fts_{/\simeq^d_{FTS}}) = [\![d]\!]_{FD}$.*

Proof. *The proof consists in showing that $\forall s \bullet \mathcal{R}_{FTS}(s, [s]_{/\simeq^d_{FTS}}) \wedge \mathcal{R}_{FTS}([s]_{/\simeq^d_{FTS}}, s) = [\![d]\!]_{FD}$. Obviously, $\mathcal{R}_{FTS}(s, [s]_{/\simeq^d_{FTS}}) = [\![d]\!]_{FD}$ because, by definition of $fts_{/\simeq^d_{FTS}}$, any transition (s, s') in fts is simulated for all products by a transition $([s]_{/\simeq^d_{FTS}}, [s']_{/\simeq^d_{FTS}})$.*

Then, we show that any transition $([s_1]_{/\simeq^d_{FTS}}, [s'_1]_{/\simeq^d_{FTS}})$ is simulated for all products by (s_1, s'_1). Indeed, $([s_1]_{/\simeq^d_{FTS}}, [s'_1]_{/\simeq^d_{FTS}})$ exists because (s_2, s'_2) is a transition of fts, with $s_2 \in [s_1]_{/\simeq^d_{FTS}}$ and $s'_2 \in [s'_1]_{/\simeq^d_{FTS}}$. If $s_1 = s_2$, we trivially have $\mathcal{R}_{FTS}([s_1]_{/\simeq^d_{FTS}}, s_1) = [\![d]\!]_{FD}$. Otherwise, we have $\mathcal{R}_{FTS}([s_1]_{/\simeq^d_{FTS}}, s_2) = [\![d]\!]_{FD}$. Since $s_2 \in [s_1]_{/\simeq^d_{FTS}}$, we also have $\mathcal{R}_{FTS}(s_2, s_1) = [\![d]\!]_{FD}$. By transitivity of the simulation relation [13], we obtain $\mathcal{R}_{FTS}([s_1]_{/\simeq^d_{FTS}}, s_1) = [\![d]\!]_{FD}$.

This implies that the two FTS have the same product-level (un)satisfiability with regard to any fLTL formula.

In spite of its straightforward computation, this first abstraction function has shown to be inefficient when it comes to actually reducing the state-space, as we will see in Section 5. Consequently, we define more efficient abstraction methods.

4.2 Reachability-Aware Simulation Quotient

The second abstraction method is similar to the first one, but it takes into account the reachability of each state when determining the equivalence classes. It requires the computation of a function $\preccurlyeq_{Rch} \bullet S \times S \to \mathbb{B}(N)$. In simple terms, $s_1 \preccurlyeq_{Rch} s_2$ gives a feature expression satisfied by the products for which s_1 and s_2 simulate each other, while considering the reachability relation associated with s_1 and s_2 respectively. More precisely, we define the following:

- s_2 trivially simulates s_1 for products that cannot reach s_1;
- s_2 cannot simulate s_1 for products that can reach s_1 but not s_2.

According to this definition, $(s_1 \preccurlyeq_{Rch} s_2)$ is given by

$$Reach(s_1) \Rightarrow (Reach(s_2) \wedge \mathcal{R}_{_{FTS}}(s_1, s_2))$$

where $Reach(s)$ denotes a feature expression satisfied by the products that can reach state s.

Next, we define the binary relation $\simeq_{Rch} \subseteq S \times S$ such that $s_1 \simeq_{Rch} s_2$ iff $(s_1 \preccurlyeq_{Rch} s_2) \wedge (s_2 \preccurlyeq_{Rch} s_1) = [\![d]\!]_{_{FD}}$. Again, $\simeq_{Rch}$ is an equivalence relation. It is obviously reflexive and symmetric. Its transitivity can be demonstrated by first observing that

$$(\mathcal{R}_{_{FTS}}(s_1, s_2) \wedge \mathcal{R}_{_{FTS}}(s_2, s_3)) \Rightarrow \mathcal{R}_{_{FTS}}(s_1, s_3)$$

.

Theorem 16 $s_1 \simeq_{Rch} s_2 \wedge s_2 \simeq_{Rch} s_3 \Rightarrow s_1 \simeq_{Rch} s_3$

Proof. *Let us note that* $s_1 \simeq_{Rch} s_2 \wedge s_2 \simeq_{Rch} s_3$ *is equivalent to*

$$(\neg Reach(s_1) \vee (Reach(s_2) \wedge \mathcal{R}_{_{FTS}}(s_1, s2)) \wedge (\neg Reach(s_2) \vee (Reach(s_3) \wedge \mathcal{R}_{_{FTS}}(s_2, s_3))$$

which can be written as

$$(\neg Reach(s_1) \wedge (\neg Reach(s_2) \vee Reach(s_3) \wedge \mathcal{R}_{_{FTS}}(s_2, s_3)) \vee (Reach(s_2) \wedge \mathcal{R}_{_{FTS}}(s_1, s_2) \wedge Reach(s_3) \wedge \mathcal{R}_{_{FTS}}(s_2, s_3))$$

Since

$$(\neg Reach(s_1) \wedge (\neg Reach(s_2) \vee Reach(s_3) \wedge \mathcal{R}_{_{FTS}}(s_2, s_3)) \Rightarrow \neg Reach(s_1)$$

and

$$(Reach(s_2) \wedge \mathcal{R}_{_{FTS}}(s_1, s_2) \wedge Reach(s_3) \wedge \mathcal{R}_{_{FTS}}(s_2, s_3)) \Rightarrow Reach(s_3) \wedge \mathcal{R}_{_{FTS}}(s_1, s_3),$$

this expression implies $s_1 \simeq_{Rch} s_3$.

Consequently, the state space of an FTS can be partitioned into equivalent classes under $\simeq_{Rch}$. Using this binary relation, we define an abstraction function that merges the states of an FTS according to their equivalence class under $\simeq_{Rch}$. Hence, the results of applying the function on an FTS fts is an abstracted FTS $fts_{/\simeq_{Rch}}$, which is defined similarly to $fts_{/\simeq^{d}_{FTS}}$.

We can show that this abstracted FTS has exactly the same behaviours as the original one, that is $fts \simeq_{FTS} fts_{/\simeq_{Rch}}$. Let $p \in [\![d]\!]_{FD}$ be a product. If $s_1 \simeq_{Rch} s_2$, then p can reach either both s_1 and s_2 or none of them:

1. If p can be reached by both s_1 and s_2, it means that it has exactly the same behavioural options in s_1 and s_2 by definition of $\simeq_{Rch}$ and $\mathcal{R}_{FTS}$. Therefore, merging s_1 and s_2 would not add any behaviour to p.

2. If p can reach neither s_1 nor s_2, then merging the two would not actually add behaviour to p since it would not be able to reach the resulting abstracted state anyway.

Theorem 17 *Let fts be an FTS. Then, we have $(fts \simeq_{FTS} fts_{/\simeq_{Rch}}) = [\![d]\!]_{FD}$.*

Proof. *By definition of $fts_{/\simeq_{Rch}}$, we trivially have $(fts \preccurlyeq_{FTS} fts_{/\simeq_{Rch}}) = [\![d]\!]_{FD}$. Next, we show that $(fts_{/\simeq_{Rch}} \preccurlyeq_{FTS} fts) = [\![d]\!]_{FD}$. For this purpose, we show that any merging does not add behaviour to fts.*

Let s_1 and s_2 such that $s_1 \simeq_{Rch} s_2$. Then, they are both in the same equivalence class. Without loss of generality, we suppose that $\{s_1, s_2\} = [s_1]_{\simeq_{Rch}}$ and that the abstraction merges only s_1 and s_2. For any product p, p can reach either both s_1 and s_2 or none of them, by definition of $\simeq_{Rch}$.

If p can reach neither s_1 or s_2, it means that any execution trace $s'_0, s'_1, \ldots$ of $fts_{|p}$ is such that $s'_i \neq s_1$ and $s'_i \neq s_2$. Consequently, because of the definition of $fts_{/\simeq_{Rch}}$, any execution trace $s'_0, s'_1, \ldots$ of $(fts_{/\simeq_{Rch}})_{|p}$ is such that $s'_i \neq [s_1]_{\simeq_{Rch}}$. Hence, no behaviour has been added.

If p can reach both s_1 and s_2, we have to show that any transition leaving $[s_1]_{\simeq_{Rch}}$ is simulated for all products by a transition of either s_1 or s_2. Note that in this case, we have $\mathcal{R}_{FTS}(s_1, s_2) = \mathcal{R}_{FTS}(s_2, s_1) = \mathbb{B}(d)$. Let s'_2 such that (s_2, s'_2) is a transition of fts. Because of the merging of s_1 and s_2 into $[s_1]_{\simeq_{Rch}}$, there is a transition $([s_1]_{\simeq_{Rch}}, s'_2)$ in $fts_{/\simeq_{Rch}}$. Thus, for any execution traces $s'_0, \ldots, s_1, \ldots$ and $s''_0, \ldots, s_2, s''_k, \ldots$ in fts, there is an execution trace $s'_0, \ldots, [s_1]_{\simeq_{Rch}}, s''_k, \ldots$, in $fts_{/\simeq_{Rch}}$. However, since s_1 and s_2 are simulation equivalent and $L(s_1) = L([s_1]_{\simeq_{Rch}})$, there is an execution trace starting from s_1 that is equivalent (in terms of successive sets of satisfied atomic propositions) to $[s_1]_{\simeq_{Rch}}, s''_k, \ldots$, in $fts_{/\simeq_{Rch}}$. Consequently, fts can reproduce any behaviour of $fts_{/\simeq_{Rch}}$.

Thus, for any fLTL property Ψ, $(fts \models_{FTS} \Psi) = (fts_{/\simeq_{Rch}} \models_{FTS} \Psi)$.

4.3 Reachability-Aware Preorder-Based Abstraction

Unlike the previous ones, the last abstraction actually modifies the behaviour of the FTS on which it is applied. Although it preserves the existing behaviour, it may add some. Informally, for any couple of states (s_1, s_2), if $(s_1 \preccurlyeq_{Rch} s_2) = [\![d]\!]_{FD}$, then s_1 is integrated into s_2. By integration, we mean that all the transitions going to s_1 are redirected to s_2 and s_2 as well as its outgoing transitions are discarded. Since s_2 simulates s_1 for any product, we do not remove any behaviour from the FTS. However, new behavioural options may appear for products in $[\![d]\!]_{FD} \cap [\![\neg(s_2 \preccurlyeq_{Rch} s_1)]\!]$.

Theorem 18 *Let s, s' be states of an FTS fts such that $(s_1 \preccurlyeq_{Rch} s_2) = [\![d]\!]_{FD}$ and $[\![\neg(s_2 \preccurlyeq_{Rch} s_1)]\!] \neq \emptyset$. Then, integrating s_1 into s_2 results in an FTS fts' such that $(fts' \preccurlyeq_{FTS} fts) \not\supseteq [\![\neg(s_2 \preccurlyeq_{Rch} s_1)]\!]$.*

Proof. *Let $p \in [\![\neg(s_2 \preccurlyeq_{Rch} s_1)]\!]$. Then, either p cannot reach s_1 in fts, or p can reach both s_1 and s_2 but there exists an execution trace starting from s_2 such that there is no execution trace starting from s_1 that is equivalent (with respect to the successive sets of satisfied atomic propositions).*

In the former case, p has no more behaviour in fts' than in fts, since any execution trace of the form $s'_0, \ldots, s'_k, s_2, \ldots$ of fts' already exists in fts or is not executable by p. In the latter case, then p has more behaviour in fts' than in fts. Indeed, let $s_2, s'_0, s'_1, \ldots$ be an execution trace starting from s_2 such that there is no equivalent execution trace starting from s_1. Therefore, for any execution trace $i, \ldots, s_1, \ldots$ of fts, with i being is an initial state of fts, there is an execution trace of the form $i, \ldots, s_2, s'_0, s'_1, \ldots$ in fts' such that there exists no equivalent execution trace in fts.

Let us observe that this form of abstraction is not a function, since for a given FTS it may lead to several abstracted FTS. For example, let s_1, s_2, s_3 be three states such that $(s_1 \preccurlyeq_{Rch} s_2) = (s_1 \preccurlyeq_{Rch} s_3) = [\![d]\!]_{FD}$, $(s_2 \preccurlyeq_{Rch} s_3) = (s_3 \preccurlyeq_{Rch} s_2) \neq [\![d]\!]_{FD}$ and there exists no s_4 such that $(s_2 \preccurlyeq_{Rch} s_4) = (s_3 \preccurlyeq_{Rch} s_4) = [\![d]\!]_{FD}$. This implies that s_1 can be integrated into either s_2 and s_3, but these two will never be merged.

Instead of defining formally the set of FTS that can result from one of these abstractions, we give an algorithm to greedily build one of its element (see Algorithm 1). First, we register the couples of states (s_1, s_2) such that $(s_1 \preccurlyeq_{Rch} s_2) = [\![d]\!]_{FD}$ in a set R (line 1). Next, we keep merging states as much as possible (lines 2-14). At each iteration, we remove an element of R (and S') non-deterministically (lines 3-4). Let (s_1, s_2) be this element. Then, s_1 is not part of S', the state-space of the abstract FTS (line 5). Furthermore, if s_1 was an initial state, then s_2 becomes an initial state of the abstract FTS (lines 6-8). As mentioned earlier, each transition of the form $(s, s_1), s \neq s_1$, is transformed into a transition (s, s_2) and the new transition-labelling function γ' is modified accordingly (lines 9-13).

Algorithm 1 Computation of the simulation function

Require: An FTS $(S, trans, I, AP, L, d, \gamma)$.

Ensure: An abstract FTS $\widehat{fts}$ smaller than fts and such that $(fts \preccurlyeq_{FTS} \widehat{fts}) = [\![d]\!]_{FD}$.

$R \leftarrow \{(s_1, s_2) : s_1 \neq s_2 \wedge (s_1 \preccurlyeq_{Rch} s_2) = [\![d]\!]_{FD}\}$
while $R \neq \emptyset$ **do**
  Let $(s_1, s_2) \in R$
  $R \leftarrow R \setminus \{(s, s') \in R \mid s = s_1 \vee s' = s_1\}$
  $S = S \setminus \{s_1\}$
  if $s_1 \in I$ **then**
    $I \leftarrow (I \cup \{s_2\}) \setminus \{s_1\}$
  end if
  $remove \leftarrow \{(s, s') \in trans \mid s = s_1 \vee s' = s_1\}$
  $trans \leftarrow (trans \setminus remove) \cup \{(s, s_2) \mid s \neq s_1 \wedge (s, s_1) \in remove\}$
  for all $s : \{(s, s_1), (s, s_2)\} \subseteq trans$ **do**
    $\gamma'(s, s_2) \leftarrow \gamma(s, s_1) \vee \gamma(s, s_2)$
  end for
end while
return $(S, trans, I, AP, L, d, \gamma')$

5 Evaluation

This section describes a theoretical evaluation of the algorithms as well as experiments we conducted to evaluate the time and space gain obtained thanks to the abstraction methods.

5.1 Theoretical Evaluation

At the heart of our method is the computation of the simulation function, as specified in Section 3.2.

Theorem 19 *The time complexity of computing the simulation function is bounded by $\mathcal{O}(|S|^6.2^{3n})$, where n is the number of features.*

Let k be the smallest such that $\forall j > k \bullet \mathcal{R}_k = \mathcal{R}_j$. For $i < k$, there is at least one triplet $(s_1, s_2, p) \in S \times S \times [\![d]\!]_{FD}$ such that $p \in [\![\mathcal{R}_{i+1}(s_1, s_2)]\!] \setminus [\![\mathcal{R}_i(s_1, s_2)]\!]$. Consequently, $k \leq |S|^2.2^n$. Assume we represent each $\mathcal{R}(s_1, s_2)$ by a Binary Decision Diagram (BDD). It is at most of size $O(2^n)$. Computing $\mathcal{R}_{i+1}$ is a conjunction or disjunction on pairs of transitions. These operations are quadratic in the size of the BDD. Thus, each step takes $|trans|^2.2^{2n} \leq |S|^4.2^{2n}$.

To verify if the fixed point has been reached, we must determine if, for all $(s_1, s_2) \in S \times S$, $\mathcal{R}_i(s_1, s_2) \leq \mathcal{R}_{i+1}(s_1, s_2)$. Establishing this comes to checking if $\mathcal{R}_i(s_1, s_2) \wedge \neg\mathcal{R}_{i+1}(s_1, s_2)$ is unsatisfiable, which is of cost $|S|^2.2^{2n}$.

	Def. 9	Def. 10
$m_{base} \preccurlyeq_{FTS} m_{ext}$	3247.07	113.39
$m_{ext} \preccurlyeq_{FTS} m_{base}$	3150.97	108.59
Total	6398.04	221.98

Table 1: Verification time of the simulation relation (in seconds)

Consequently, the overall time complexity of computing $\mathcal{R}_{FTS}$ is bounded by $\mathcal{O}(|S|^6.2^{3n})$. Although it is theoretically dominated by 2^{3n}, and thus in EXPTIME, in practice $|S|^6$ is often bigger.

5.2 Evaluation of Simulation-based Verification

To carry out these experiments, we have integrated the computation of the simulation function as well as the three abstractions into our Haskell FTS library[1], which we previously used for benchmarking our LTL model-checking algorithms [5]. It allows us to validate our approach and to measure its efficiency when it comes to computing the simulation relation and reducing both the state-space size of an FTS and its verification time. All benchmarks were run on a MacBook Pro with a 2,4 GHz Core 2 Duo processor and 4 Gb of RAM. The library was compiled using the Glasgow Haskell Compiler[2]. To avoid the influence of other running processes, we repeated each experiment 10 times.

Our evaluation considers the mine pump controller defined in [16], which we already used in our previous work [5]. The whole system is designed as the parallel composition of several processes (a pump, a water sensor, a methane sensor, and a controller). The mine pump SPL has nine features and 64 products. The FTS modelling its behaviour, noted m_{base} is composed of 465 states and 1306 transitions (see [7] for a detailed description).

In Section 3, we introduced two methods for computing the simulation relation for two FTS. The former is based on an enumerative approach and determine, for given fts_1 and fts_2, and each product p, if $fts_{1\,|p} \preccurlyeq_{TS} fts_{2\,|p}$. The latter makes use of the compact structure of FTS and is based on the computation of a fixed point, as stated in Subsection 3.2. Our first experiments evaluate the practical efficiency of both methods.

The evaluation considers the minepump system, m_{base}, as well as an extension of it. Basically, we extended the behavioural options of some of the products by making them able to execute additional transitions. This results in an extended model, noted m_{ext}. Then, we measured the time needed by both methods to compute $m_{base} \preccurlyeq_{FTS} m_{ext}$ and $m_{ext} \preccurlyeq_{FTS} m_{base}$. Benchmarks results are shown in Table 1.

[1] `http://info.fundp.ac.be/~acs/fts/implementations/haskell-library/`

[2] `http://www.haskell.org/ghc/`

We observe that the algorithm based on Definition 10 is far more efficient than the one that enumerates the products and computes the TS-simulation of their projection. In spite of having a worse theoretical time complexity, it is 28.82 times faster than the enumerative algorithm. Note that the execution time of the enumerative algorithm includes the time needed for determining the projection of each product, which amounts to about 20% of the whole execution time.

5.3 Evaluation of Temporal Property Verification

Enumerative method

Formula	m_{base}		$m_{\simeq_{TS}}$		$m_{\preccurlyeq_{TS}}$	
#1	31.23	✗	40.42	✗	39.70	✗
#2	9.79	✓	19.58	✓	19.75	✓
#3	134.71	✗	178.52	✗	175.11	✗
#4	8.38	✓	17.2	✓	17.97	✓
#5	19.18	✓	32.22	✓	33.51	✗
#6	27.57	✗	37.94	✗	36.69	✗
#7	9.44	✓	19.37	✓	20.26	✓
#8	46.19	✗	64.48	✗	62.22	✗
#9	11.39	✗	24.58	✗	25.64	✗
#10	8.78	✓	19.54	✓	19.25	✓
Total	**306.66**		**453.85**		**450.1**	

FTS algorithms

Formula	m_{base}		$m_{\simeq_{Rch}}$		$m_{\preccurlyeq_{FTS}}$	
#1	13.08	✗	13.46	✗	12.07	✗
#2	0.81	✓	2.15	✓	1.94	✓
#3	97.91	✗	92.37	✗	82.23	✗
#4	0.96	✓	2.31	✓	2.01	✓
#5	1.26	✓	2.56	✓	2.29	✗
#6	10.10	✗	10.89	✗	9.91	✗
#7	0.41	✓	1.79	✓	1.62	✓
#8	7.2	✗	7.7	✗	6.80	✗
#9	0.65	✗	2.02	✗	1.79	✗
#10	0.49	✓	1.86	✓	1.67	✓
Total	132.87		137.11		122.33	

Table 2: Verification time of ten LTL formulae (in seconds)

Our second evaluation benchmarks the time needed for model checking

abstractions combined with either the enumerative approach or FTS algorithms. The objective of the following experiments is to determine which abstraction method is more efficient. For the former method, we evaluate the verification time by enumerating all the products and computing their projection on (1) the original model, (2) its TS-simulation quotient $m_{\simeq_{TS}}$ [13], and (3) the model $m_{\preccurlyeq_{TS}}$, obtained by integrating a state s_1 into another s_2 iff $s_1 \preccurlyeq_{TS} s_2$.

Also, we apply the aforementioned abstractions to obtain three abstract FTS $m_{\simeq_{FTS}}$, $m_{\simeq_{Rch}}$, and $m_{\preccurlyeq_{FTS}}$ respectively. For the latter, when a state can be integrated into more than one state, our choice is based on the lexicographic order. For example, if we have $s_1 \preccurlyeq_{FTS} s_2 = s_1 \preccurlyeq_{FTS} s_3 = \mathbb{B}(d)$ then we integrate s_1 into s_2 rather than in s_3.

For each FTS, we first compute the number of states and transitions in order to determine to what extent a given abstraction reduces the size of the original FTS. We observe that the abstraction based on the equivalence classes under $\simeq_{FTS}$ yields no reduction at all. Its merging condition is too restrictive in the context of product lines. Since an FTS models the behaviour of $\mathcal{O}(2^n)$ products, it is very unlikely that two states have exactly the same behavioural options for all those products.

Taking into account the reachability already allows to merge states, although only a few of them. The state-space size is thus reduced to 459 states and 1284 transitions. Finally, the third abstraction yields a reduction of about 9% (423 states and 1192 transitions). Although these are the best results in terms of state-space reduction, we must keep in mind that, like every efficient state-space reduction method, it augments the behaviour of the FTS. Hence, we may find false negatives, *i.e.* products that violate a given property in the abstract FTS but not in the original one.

In order to evaluate the impact of the state-space reduction on the verification time, we model-checked the seven models against ten different properties expressed in LTL, such as those defined in [17] and [5]. The results are shown in Table 2. For every formula and every model, we give the time needed to verify the model against the formula. We also describe if the formula is verified by every legal product (✓) or not (✗). The verification time of every property includes the computation time of the abstractions, which represents about 10% of the overall verification time, in both cases. This overhead could be partially avoided if the abstractions are computed once and for all. We do not present the verification times for $m_{\simeq_{FTS}}$. Since it has as many states and transitions as m_{base}, any difference would be the result of random variations independent of the verification process.

Let us first discuss the results for the enumerative and FTS approaches separately. When summing up all the times related to the enumerative approach, we observe that the overhead due to the computation of both the abstractions based on TS-simulation quotient ($m_{\simeq_{TS}}$) and preorder ($m_{\preccurlyeq_{TS}}$) is significant. Because of that, the verification times of $m_{\simeq_{TS}}$ and $m_{\preccurlyeq_{TS}}$

are respectively 48% and 47% higher than the model-checking time of m_{base} using the enumerative method. Furthermore, false negatives appeared when checking $m_{\preccurlyeq_{TS}}$ against formula #5. This formula is supposed to be satisfied by all products, but one of them violates it in $m_{\preccurlyeq_{TS}}$ due to the addition of behaviour.

Applying abstraction to FTS yields better results. The abstraction under $\simeq_{Rch}$ increases the checking time of m_{base} with FTS by 3%. This verification time decreases by 8% when $m_{\preccurlyeq_{FTS}}$ is model-checked. However, it has to be noted that false negatives were found for formula #5. If we compare these results with the ones of the enumerative methods, we conclude that the FTS-based approaches outperform the enumerative ones. Furthermore, applying abstraction to an FTS can reduce its verification cost. On the contrary, combining an enumerative method with an abstraction function is inefficient.

Although abstraction clearly permits to reduce the verification of an FTS, we are aware that the gain is not significant. This illustrates the difficulty to find a good abstraction for FTS, a formalism that models the behaviours of a potentially large number of systems. A good abstraction should either add behaviour or remove some, but not both. Otherwise, we would not be able to infer any property of the system using the verification results of its abstract counterpart, since a property may be violated by an additional behaviour or satisfied thanks to the removal of an existing one. Therefore, it is particularly difficult to find a state merging condition that both satisfies this requirement and makes a significant reduction. In particular, the purpose of simulation quotient is to eliminate redundancy, not to produce coarse abstractions. More research is required to find ways to design efficient abstraction functions and to finely evaluate their merits with respect to verification performance and false negatives induced by them. The current abstractions are applied directly on the FTS itself. The most successful applications of abstraction, like partial-order reduction and statement merging, make use of additional information like parallelism and variables scope. These information are not found in such a fundamental formalism, but instead in high-level languages.

Nevertheless, it is interesting to observe that the most important speedups occur during the verification of the most time-consuming properties. This indicates that abstraction can play a role in improving the scalability of SPL verification. Naturally, this early indication needs to be confirmed by further experiments. These results combined with previous experiments [5–7] confirm that FTS is a viable approach for verifying variability-intensive systems.

5.4 Threats to Validity

Several threats to the validity of our conclusions have to be pointed out. First, our evaluation is solely based on one case. Other systems of different size and variability should be considered in order to analyse how the different approaches scale with the number of features and the size of the state-space.

This case study has been implemented in Haskell, a programming language that makes use of the so-called lazy evaluation. It means that a value is computed only when it is needed. Although this evaluation method may have influenced our results, the conclusions would certainly remain valid if we used another programming language.

The comparison between the enumerative methods and the FTS algorithms is based on the verification of all the products. However, when a property (or a simulation relation) is required to hold for the whole SPL, we could stop the checking process as soon as a bad product is found. Even so, this comes to the standard model checking problem, and we are interested in identifying all the products that violate a property.

Also, we obtain the verification times related to the enumerative methods by summing up the verification times for each product individually. In practice, it is very unlikely that those products are verified sequentially, without taking advantage of multi-threading and parallel verification.

Finally, independent processes running during the experiments might have influenced the results. However, each experiment has been repeated 10 times. This way, the impact of those random variations is drastically reduced.

6 Related work

This section briefly describes relevant work related to modelling and verification of SPL behaviour.

Fischbein *et al.* propose Modal Transition Systems (MTS) to model the behaviour of SPLs [18]. An MTS is a TS where transitions are either mandatory or optional. The mandatory transitions are available to all products whereas optional ones are specific. Although model checking an MTS determines if a property is satisfied by all or only a subset of the products, it does not keep track of the decisions made at variation points and it lacks the notion of feature. Therefore, it cannot pinpoint exactly the products that violate the property. Asirelli *et al.* [19] associate MTS with the MHML temporal logic [20] to express constraints on features . Still, since they do not have an explicit notion of feature *in* the MTS, they suffer from the same limitations.

Sassolas *et al.* [21] propose a method to identify inconsistencies between several MTS based on traces comparison and a simulation relation. The inconsistencies are characterised as μ-calculus formulae. Unlike ours, their

approach is not specific to SPL and cannot be used to identify products that cause inconsistencies.

Instead of MTS, Larsen *et al.* apply I/O automata to SPL modelling [22]. In particular, they define an SPL as the composition of subfamily modelled with an I/O automata. However, they do not address SPL verification.

Lauenroth *et al.* define a CTL model-checking algorithm for automata labelled with features [23]. There are two significant differences between their work and ours. First, they do not allow to label transitions with any arbitrary boolean expressions. Second, the time complexity of their algorithms are exponential in the state-space size and they have not applied state-space reduction techniques.

Ghezzi and Molzam Sharifloo verify non-functional properties (reliability, energy consumption, ...) in SPLs with probabilistic model-checking [24]. Their work is complementary to ours but does not rely on a formal model. Transposing our approach to probabilistic model checking is a promising research perspective.

Cassez *et al.* [25] make use of the simulation relation for alternating-time temporal logic (ATL) to prove the non-interaction of features in reactive systems. They establish syntactic conditions for a feature to preserve properties. Similarly, Fisler *et al.* [26], Krishnamurthi *et al.* [27] and Li *et al.* [28] introduce an approach for compositional model-checking of collaborations, aspects and features. Both the base system (i.e. the system without features) and the features are modelled as a finite state machine (FSM). Enabling the feature means attaching its FSM to the one of the base system. They propose algorithms that derive preservation constraints which, if satisfied by the feature FSM, ensure that a given CTL formula verified in the base system is also satisfied when the feature is enabled. One limitation is that their features only *add* transitions and states. In the same vein, Liu *et al.* [29] propose an alternate algorithm to derive the preservation constraints. Transposed to FTS, these ideas could open a way for compositional verification of SPL.

7 Conclusion

In this paper, we focused on providing theoretical foundations and empirical evidence to apply simulation-based model checking to SPLs. First, we defined a simulation relation for FTS, a formalism meant to model the behaviour of all the products of a SPL. Simulation relations add a significant milestone to SPL verification theory, being at the center of advanced behavioural analyses, such as abstraction, behavioural comparison, compositional reasoning, and more. The second contribution is the study of simulation quotients for FTS, which results in several simulation-based abstraction methods. The third contribution is the evaluation of these abstractions

for SPL model checking. The main conclusion of our experiments is that the combination of abstraction with the enumerative approach is inefficient. This corroborates the claims we made in earlier work that SPL model checking should be based on FTS. However, our experiments also suggest that the application of abstraction to FTS model checking only yields marginal efficiency gains. To obtain more substantial improvements, our approach should be extended with other abstraction methods. Counter-example guided abstraction refinement (CEGAR) [30] looks particularly promising: a coarse abstraction is rapidly and automatically computed and is then refined iteratively using the false negatives found during verification.

As other future work, we plan to integrate our results in an SPL model-checking tool equipped with a high-level specification language, *viz.* SNIP [7]. This would allow us to apply other forms of abstraction, (*e.g.*, partial-order reduction [2, 10, 13] and statement merging [10]), and to evaluate them on larger models, including industrial cases. However, it requires to extend definitions such as stutter equivalence [13, 31] to variability-intensive models.

Apart from abstraction, there are many other uses of the simulation function we have defined. For instance, simulation-based verification allows one to verify properties modelled as automata. There are also SPL-specific uses of our theory. For instance, we can formally characterise the behavioural impact of features in an SPL. Let f be a feature, $fts^{[f]}$ (resp. $fts^{[\neg f]}$) the FTS modelling the behaviour of the products that have (resp. do not have) f. Then, ($fts^{[\neg f]} \preccurlyeq_{FTS} fts^{[f]}$) gives the products for which f does not remove existing behavioural options. Inter-SPL comparison is also possible. If we consider two SPLs having an equal set of legal products, the behavioural inconsistencies between them can be highlighted and presented in the form of an automata (*viz.* an FTS). A similar approach for modal transition systems is studied by Sassolas *et al.* [21].

Moreover, we plan to investigate the use of the simulation for compositional reasoning and verification of variability-intensive systems. Simulation relations are already at the core of existing research on compositional verification, in particular for discrete and hybrid systems [32]. Applying similar methods to verify behavioural variability models compositionally is an exciting but difficult challenge, considering the numerous possible interactions between features.

Finally, we will extend the above results as well as our previous work on FTS to the modelling and verification of variability-intensive real-time systems. Analysing the behaviour of such systems requires (1) the definition of models that combine FTS with timed automata, (2) the development of model checking algorithms for verifying time-critical properties on these models, and (3) the definition of timed simulation for FTS augmented with real-time. More generally, this work is part of a larger project that aims to extend the theory, methodologies and tools for the behavioural modelling and verification of SPL.

References

[1] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps, "Feature Diagrams: A Survey and A Formal Semantics," in *RE'06*, 2006, pp. 139–148.

[2] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[3] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *LICS'86*. IEEE CS, 1986, pp. 332–344.

[4] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, pp. 244–263, April 1986.

[5] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," in *Proceedings of ICSE 32*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 335–344.

[6] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic model checking of software product lines," in *ICSE 33*. ACM, 2011, pp. 321–330.

[7] A. Classen, "Modelling and model checking variability-intensive systems," Ph.D. dissertation, 2011.

[8] A. Pnueli, "The temporal logic of programs," in *Proc. 18th Annual Symposium on Foundations of Computer Science (FOCS)*, 1977, pp. 46–57.

[9] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of Programs*, ser. LNCS, vol. 131. Springer, 1981, pp. 52–71.

[10] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[11] G. Bruns, "A practical technique for process abstraction," in *Proceedings of the 4th International Conference on Concurrency Theory*, ser. CONCUR '93. London, UK: Springer-Verlag, 1993, pp. 37–49.

[12] R. Milner, "An algebraic definition of simulation between programs," Stanford University, Stanford, CA, USA, Tech. Rep., 1971.

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

[14] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," SEI, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[15] [Online]. Available: http://info.fundp.ac.be/~acs/snip

[16] J. Kramer, J. Magee, M. Sloman, and A. Lister, "Conic: an integrated approach to distributed computer control systems," *Computers and Digital Techniques, IEE Proceedings E*, vol. 130, no. 1, pp. 1–10, 1983.

[17] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel, "Learning operational requirements from goal models," in *ICSE 31*, 2009, pp. 265–275.

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

[19] P. Asirelli, M. H. T. Beek, A. Fantechi, and S. Gnesi, "Formal description of variability in product families," in *Proceedings of the 15th International Software Product Line Conference*, ser. SPLC'11. Springer-Verlag, 2011, pp. 130–139.

[20] ——, "A logical framework to deal with variability," in *Proceedings of the 8th international conference on Integrated formal methods*, ser. IFM'10. Berlin, Heidelberg: IEEE, 2010, pp. 43–58.

[21] M. Sassolas, M. Chechik, and S. Uchitel, "Exploring inconsistencies between modal transition systems," *Softw. Syst. Model.*, vol. 10, pp. 117–142, February 2011.

[22] K. G. Larsen, U. Nyman, and A. Wasowski, "Modal I/O automata for interface and product line theories," in *ESOP*, 2007, pp. 64–79.

[23] K. Lauenroth, S. Thning, and K. Pohl, "Model checking of domain artifacts in product line engineering," in *IEEE/ACM ASE*, 2009, pp. 269–280.

[24] C. Ghezzi and A. Molzam Sharifloo, "Verifying non-functional properties of software product lines: Towards an efficient approach using parametric model checking," in *Proceedings of the 15th International Software Product Line Conference*, ser. SPLC'11. Springer-Verlag, 2011, pp. 170–174.

[25] L. Brim, I. Cerna, P. Krcál, and R. Pelánek, "Distributed ltl model checking based on negative cycle detection," in *Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science*, ser. FST TCS '01. London, UK: Springer-Verlag, 2001, pp. 96–107.

[26] K. Fisler and S. Krishnamurthi, "Modular verification of collaboration-based software designs," in *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-9. New York, NY, USA: ACM, 2001, pp. 152–163.

[27] S. Krishnamurthi, K. Fisler, and M. Greenberg, "Verifying aspect advice modularly," in *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering.* New York, NY, USA: ACM, 2004, pp. 137–146.

[28] H. C. Li, S. Krishnamurthi, and K. Fisler, "Interfaces for modular feature verification," in *ASE*, 2002, pp. 195–204.

[29] J. Liu, S. Basu, and R. R. Lutz, "Compositional model checking of software product lines using variation point obligations," *Automated Software Engg.*, vol. 18, pp. 39–76, March 2011.

[30] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, E. Emerson and A. Sistla, Eds. Springer Berlin / Heidelberg, 2000, vol. 1855, pp. 154–169.

[31] J. F. Groote and F. Va, "An efficient algorithm for branching bisimulation and stuttering equivalence," in *Proceedings of the seventeenth international colloquium on Automata, languages and programming.* Springer-Verlag, 1990, pp. 626–638.

[32] G. Frehse, "Compositional verification of hybrid systems using simulation relations," Ph.D. dissertation, Radboud Universiteit Nijmegen, 2005.