Published in: ***Proceedings of the International Software Product Line Conference (SPLC'16),*** September 2016

"Long-term Average Cost in Featured Transition Systems"

Cite as:

Rafael Olaechea, Uli Fahrenberg, Joanne M. Atlee, and Axel Legay. 2016. Long-term average cost in featured transition systems. In Proceedings of the 20th International Systems and Software Product Line Conference (SPLC '16). ACM, New York, NY, USA, 109-118. DOI: https://doi.org/10.1145/2934466.2934473

BibTex:


```
@inproceedings{Olaechea:2016:LAC:2934466.2934473,
 author = {Olaechea, Rafael and Fahrenberg, Uli and Atlee, Joanne M. and Legay, Axel},
 title = {Long-term Average Cost in Featured Transition Systems},
 booktitle = {Proceedings of the 20th International Systems and Software Product Line
Conference},
 series = {SPLC '16},
 year = {2016},
 pages = {109--118}
}
```


DOI: https://doi.org/10.1145/2934466.2934473

Long-Term Average Cost in Featured Transition Systems

Rafael Olaechea, Uli Fahrenberg, Joanne M. Atlee, Axel Legay
University of Waterloo, Canada
Inria Rennes, France

ABSTRACT

A software product line is a family of software products that share a common set of mandatory features and whose individual products are differentiated by their variable (optional or alternative) features. Family-based analysis of software product lines takes as input a single model of a complete product line and analyzes all its products at the same time. As the number of products in a software product line may be large, this is generally preferable to analyzing each product on its own. Family-based analysis, however, requires that standard algorithms be adapted to accomodate variability.

In this paper we adapt the standard algorithm for computing limit average cost of a weighted transition system to software product lines. Limit average is a useful and popular measure for the long-term average behavior of a quality attribute such as performance or energy consumption, but has hitherto not been available for family-based analysis of software product lines. Our algorithm operates on weighted featured transition systems, at a symbolic level, and computes limit average cost for all products in a software product line at the same time. We have implemented the algorithm and evaluated it on several examples.

1. INTRODUCTION

Many of today's software-intensive systems are developed as a family of related systems (e.g., smart phones, automotive software). In particular, a *software product line (SPL)* is a family of software products that share a common set of mandatory features and whose individual *products* are differentiated by their variable (optional or alternative) *features*.

Analysis of software product lines can be categorized into family-based or product-based [22]. Product-based techniques analyze each possible product (or a sample subset of products) individually, whereas a family-based analysis is performed on a single model that represents all of the products in an SPL. Thus, family-based approaches avoid some of the redundant computations inherent in product-based analyses; but they require that standard analysis algorithms be adapted to accommodate variability in the SPL model.

Quality attributes of software systems (e.g., performance, energy consumption) are a key concern when developing and evaluating software products. An especially useful analysis of quality attributes, called *limit average*, computes a long-term average of a quality attribute for a product. In this paper, we adapt this algorithm to perform a family-based analysis that computes at once the limit average for all products in a software product line.

Our contributions include:

- A family-based algorithm that analyzes a model of an SPL and computes the maximum limit average for a quality attribute, for all products at the same time.
- An implementation of the family-based algorithm.
- An evaluation of the speed-up of our family-based approach versus the product-based approach.

Due to space constraints, we display some of our algorithms in a separate appendix; but we give short explanations of their working in the paper itself.

1.1 Related Work

Product Line Analysis.

Lauenroth *et al.* [20] introduce an algorithm to verify a product line, represented as an I/O automaton with optional transitions annotated with features, against properties expressed in computational tree logic (CTL). Their algorithm checks that every possible I/O automaton that can be derived satisfies a given CTL property. Lauenroth *et al.* mention that CTL properties of a special form can be checked by restricting the automaton and checking if all non-trivial strongly connected components (SCCs) of this restricted automaton can be reached from the initial state. They then adapt this algorithm by replacing the computation of SCCs with a procedure to find a path to a cycle, keeping track of the features required along such a path to a cycle. In our case we are instead interested in finding all the products for which each cycle with a given average cost is reachable. Lauenroth *et al.* do not compare the performance of their family-based approach with respect to a product-based approach.

Classen *et al.* [7] adapt the standard algorithm for model checking properties of transition systems expressed in linear temporal logic (LTL) to analyze a product line represented as a featured transition system. Their approach is between

2 and 38 times faster than analyzing each product individually. Although they represent products symbolically, they still represent the transition system using explicit states and transitions. In subsequent work, Classen *et al.* [8] extend their approach to transition systems represented symbolically. They adapt the algorithm for model checking CTL properties to a family based approach and show speed-ups of several orders of magnitude versus verifying each product individually. Hence both LTL and CTL model checking have been adapted to analyze a family of transition systems instead of individual products, showing orders of magnitude speed-ups compared to analyzing each product individually.

More recently, Ben-David *et al.* [1] have adapted SAT-based model checking of safety properties to a family based approach and showed that such approach was substantially faster than the methods by Classen *et al.*

Limit-Average Cost.

Quantitative methods are important in performance analysis [17], reliability analysis [21], and other areas of software engineering. Long-term average values are often used, for example to measure mean time between failures or average power consumption; see also [13, 15] for further motivation.

In [3], Černý *et al.* show how limit average cost can be used to measure the distance between a specification and an incorrect implementation. They define a limit-average correctness distance to capture how frequently the specification has to "cheat" in order to simulate the incorrect implementation. This work is generalized to interfaces and abstractions in [4, 5].

In [11, 12], Fahrenberg and Legay argue more generally for an approach of quantitative model checking which measures distances between models and specifications; a similar proposal is Henzinger and Otop's [14]. As a specific example, Boker *et al.* in [2] extend LTL with limit-average path accumulation assertions and show that model checking quantitative Kripke structures with respect to this LTL extension is decidable.

We are not aware of any family-based analysis methods which compute the limit average cost for all products in a software product line.

2. BACKGROUND

A *transition system* (TS) is composed of a set of states, actions, transitions and a set of initial states. Hence, it is a tuple $ts = (S, Act, trans, I)$, where $trans \subseteq S \times Act \times S$ and $I \subseteq S$. An *execution* of a transition system is an alternating infinite sequence of states and actions $\pi = s_0\alpha_1 s_1\alpha_2 \ldots$ with $s_0 \in I$ such that $(s_i, \alpha_{i+1}, s_{i+1}) \in trans$ for all i. The semantics of a TS (written as $[\![ts]\!]$) are given by its set of executions.

A software system may have to satisfy not only *functional requirements*, which can be expressed and verified for example through logical properties, but also *quality requirements* such as maximum energy consumption or timing constraints. Hence transition systems have been extended with weights to model these quality attributes. A *weighted transition system* is thus a tuple $wts = (S, Act, trans, I, W)$, where $(S, Act, trans, I)$ is a transition system and $W : trans \to \mathbb{R}$ is a function that assigns real weights to transitions.

2.1 Limit Average Cost

The *limit average cost* expresses the average of weights in a single infinite execution of a weighted transition system. Thus, if the weights represent the consumption of a resource, then the limit average represents the long term rate of resource consumption along a single (infinite) execution.

Given an infinite execution $\pi = s_0\alpha_1 s_1\alpha_2 \ldots$ of a weighted transition system, we define a corresponding infinite sequence of weights $w(\pi) = v_0 v_1 \ldots$ where $v_i = W(s_i\alpha_{i+1}s_{i+1})$. The limit average of π is then defined to be

$$LimAvg(\pi) = \liminf_{n\to\infty} \frac{1}{n}\sum_{i=0}^{n} v_i\,.$$

The maximum (or minimum) limit average of a weighted transition system is the maximum (or minimum) limit average over all of its execution traces. For example, by computing the minimum and maximum limit average of a weighted transition system whose weights represent energy consumption, we obtain the best-case and worst-case long-term rates of energy consumption.

Computation of maximum or minimum limit-average cost is entirely analogous. In this paper we focus on maximum limit-average cost, but everything we do can also be applied to minimum limit-average cost. The maximum limit average can be computed by a two-phase algorithm [23]: first one computes the set of strongly connected components, and then for each strongly connected component one identifies the cycle with the highest mean-weight. Finally, the mean weight of the maximum mean-weight cycle reachable from the initial state is the maximum limit average for the weighted transition system.

A strongly connected component (SCC) is a maximal set of nodes in a graph such that there exists a directed path between every pair of nodes in the set. Any cycle in a graph will be contained inside an SCC, hence by searching for maximum mean-weight cycles in each SCC of a graph we obtain the maximum mean-weight cycle of the full graph.

The standard algorithm [10] for computing the SCCs of a graph $G = (V, E)$ performs a depth-first search of the graph and computes for each node its "finishing time" in the depth-first search. The finishing time $F(v)$ of a node v represents the temporal order at which a node and all its forward neighbors have been fully explored, and ranges from 1 to $|V|$.

The algorithm for computing SCCs then processes the nodes in decreasing finishing times. It starts at the node v with $F(v) = |V|$ and computes the set of nodes that can be reached from v in the transpose of the graph (i.e. the graph that has the same nodes and edges but with reversed edge directions). These sets of nodes correspond to an SCC. The algorithm then removes this SCC from the graph and processes the remaining nodes in decreasing order of finishing times, until each node has been assigned to an SCC. The SCC algorithm takes time $O(V + E)$.

In order to compute the maximum limit-average cost, we now need to calculate the highest mean-weight cycle in each SCC. This is usually done using Karp's algorithm [19]. This algorithm choses an arbitrary initial state s_0 and then iteratively computes a function D which associates with each state v and each $k \in \{0, \ldots, n\}$, where n is the size of the SCC, the maximal weight of a path of length k from v to s_0. By Karp's theorem [19], the weight of the maximal mean-weight cycle is then given as $\max_v \min_{k<n} \frac{D[n,v]-D[k,v]}{n-k}$.

2.2 Weighted Featured Transition Systems

A *feature model* [18] is used to configure a software product line. It represents the set of valid products. For our purposes a feature model is used exclusively to distinguish between valid and invalid products, hence it is a tuple $d = (N, px)$, where N is the set of features and $px \subseteq \mathcal{P}(N)$ is the set of products. Here $\mathcal{P}(N)$ denotes the power set of N; an individual software product is thus composed of a set of features.

A transition system represents the behavior of a single software product. In order to analyze all the products of a software product line at the same time, Classen *et al.* [7] have introduced featured transition systems which compactly represent the behavior of all the products of a software product line.

A boolean feature variable represents the presence or absence of a feature in a software product. A product is then represented by an assignment of values to all feature variables (true if the feature is present in the product, false if not). Hence we can represent a set of products by a *boolean feature expression* - that is, a boolean formula over feature variables, whose solutions represent the set of products. We denote by $\mathbb{B}(N)$ the set of such feature expressions.

A *featured transition system* annotates each transition with a boolean feature expression, which corresponds to the set of products whose transition system include that transition. It is thus a tuple $fts = (S, Act, trans, I, d, \gamma)$, where $(S, Act, trans, I)$ is a transition system, $d = (N, px)$ is a feature model, and $\gamma : trans \to \mathbb{B}(N)$ labels each transition with a feature expression.

Therefore FTSs unify the transition systems of all products in a product line into a single annotated transition system. The featured transition system provides a 150% model of all products' states and transitions – that is, it includes more transitions and states than required for each individual product.

The transition system for each specific software product can be derived by removing all annotated transitions whose feature expression is not satisfied by the product's feature-variable assignment. This transition system contains all the states of the FTS and all the transitions whose feature expressions evaluate to true under the software product. Formally, the *projection* of an FTS fts to a product $p \in [\![d]\!]$, noted $fts_{|p}$, is the TS $ts = (S, Act, trans', I)$ where $trans' = \{t \in trans \mid p \vDash \gamma(t')\}$.

A featured transition system can be extended with weights on transitions in the same way that transition systems can, in which case each product of the software product line is represented by a weighted transition system. Then we can compute the maximum limit average for each product of the software product line. A *weighted featured transition system* (WFTS) is thus a tuple $wfts = (S, Act, trans, I, d, \gamma, W)$, where $(S, Act, trans, I, d, \gamma)$ is an FTS and $W : trans \to \mathbb{R}$ is a function that annotates transitions with weights.

A WFTS can be projected for a specific product into a weighted transition system, analogously to FTS projection as above: the *projection* of a WFTS $wfts$ to a product $p \in [\![d]\!]$, denoted $wfts_{|p}$, is the WTS $wts = (S, Act, trans', I, W)$ where $trans' = \{t \in trans \mid p \vDash \gamma(t)\}$.

3. MOTIVATING EXAMPLE

Figure 1 shows an (artificial) example of a combined taxi

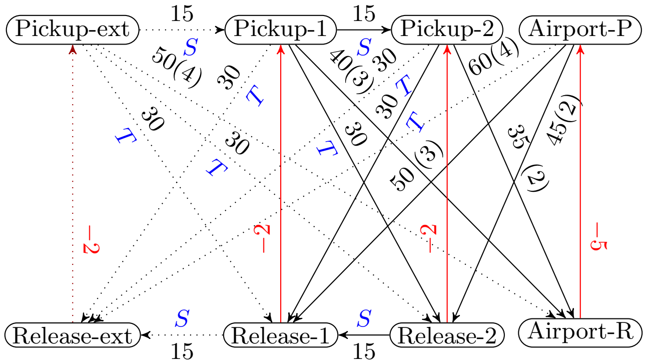

Figure 1: Taxi-shuttle example. In addition to the feature guards shown, all *dotted* transitions are guarded by the feature L.

and shuttle service. There are three pickup and release locations in the city, one of which is only available when the car has an extra license (feature L). Additionally, passengers can be picked up and released at the airport. Taxi service (feature T) is available within city locations, not just for transportation to and from the airport. Shuttle service (feature S) allows to pick up passengers at several pickup locations before delivering them to the airport, or to pickup passengers for several different city locations at the airport.

The weights on the transitions show their cost; positive numbers are income for the driver, negative numbers are expenses. To model the fact that travels to the airport take longer time than travels in the city, the transitions to and from the airport have length 2 (from the second pickup point), 3 or 4. In practice we will model this by inserting extra states and transitions of weight 0.

The example has thus three features, S, T and L, giving rise to eight products: $\emptyset$, $\{L\}$, $\{S\}$, $\{T\}$, $\{L, S\}$, $\{L, T\}$, $\{S, T\}$, and $\{L, S, T\}$. An interesting problem is to compute maximal income for the driver, depending on the product; the maximum limit average cost is a reasonable approximation of this maximal income.

A product-based analysis reveals that regardless of the feature selection, the transition system always has precisely one SCC. For the product $p = \emptyset$, there are two cycles:

$$\text{Airport-P} \to \text{Release-1} \to \text{Pickup-1} \to \to \text{Airport-R} \to \text{Airport-P} \quad (1)$$

$$\text{Airport-P} \to \text{Release-2} \to \text{Pickup-2} \to \to \text{Airport-R} \to \text{Airport-P} \quad (2)$$

Their mean weights are 10.38 and 12.17, respectively (rounded to two places), hence cycle (2) between the airport and city location 2 provides the maximal income.

For $p = \{L\}$, another cycle becomes available:

$$\text{Airport-P} \to \text{Release-ext} \to \text{Pickup-ext} \to \to \text{Airport-R} \to \text{Airport-P} \quad (3)$$

But its mean weight is only 10.30, so cycle (2) is still the most profitable.

If $p = \{S\}$, then additionally to cycles (1) and (2) above,

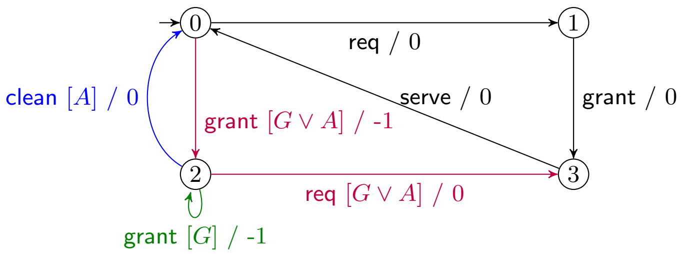

Figure 2: WFTS which implements several grant/request scenarios

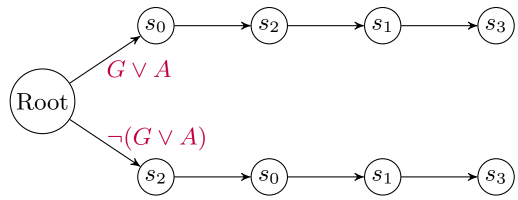

Figure 3: Symbolic finishing-times tree for the FTS from Fig. 2

three other cycles become available:

$$\text{Airport-P} \to \text{Release-2} \to \text{Release-1} \to \\ \to \text{Pickup-1} \to \text{Airport-R} \to \text{Airport-P} \quad (4)$$

$$\text{Airport-P} \to \text{Release-1} \to \text{Pickup-1} \to \\ \to \text{Pickup-2} \to \text{Airport-R} \to \text{Airport-P} \quad (5)$$

$$\text{Airport-P} \to \text{Release-2} \to \text{Release-1} \to \text{Pickup-1} \to \\ \to \text{Pickup-2} \to \text{Airport-R} \to \text{Airport-P} \quad (6)$$

Their mean weights are 11.63, 11.63, and 12.88, respectively, hence for a pure shuttle, cycle (6) which picks up and releases passengers at both city locations is most profitable.

Similar analyses can be done for the other five products, but a family-based analysis which computes SCCs and maximum mean-weight cycles for all products at once would be preferable. We will come back to this example in Section 5.

4. FAMILY-BASED LIMIT AVERAGE COMPUTATION

We want to compute the maximum limit average cost for each product in a software product line. We propose a family-based algorithm that re-uses partial computation results that apply to multiple products. The algorithm starts by computings SCCs (subsections 4.1 and 4.2) and then for each SCC it computes its maximum mean cycle (subsection 4.3).

In order to illustrate the family-based SCC computation, we introduce another example. Consider three solutions to the problem of an arbiter granting access to a shared resource, modeled as a WFTS in Fig. 2. One solution involves granting access only after a request has been received: this will be the solution implemented by the basic system without the optional features A or G. An alternative solution is to always grant access, whether a request exists or not. This is implemented by the product with feature G. A third option is to alternate between granting access and not granting access, implemented by the product with feature A.

Each of these solutions satisfies the functional requirements of the system, namely that a request is always granted. However the user may prefer one solution over another: for example she might want to minimize the number of unnecessary grants. These preferences are encoded as weights on the transitions, such that every time a grant is given when not needed, or when a request has to wait before being served, a penalty of -1 is given.

4.1 Symbolic Finishing Times

The algorithm for computing SCCs of a graph depends on the finishing times of states in a depth-first search. However a featured transition system represents a set of transition systems, each with a different set of transitions, which can give rise to a different set of depth-first finishing times for its states. For example the basic product in Fig. 2 (without feature A nor G) would have the following finishing times of states:

$$F(s_3) = 1, F(s_1) = 2, F(s_0) = 3, F(s_2) = 4\,,$$

whereas in any product that includes feature A, state s_0 has the highest finishing time:

$$F(s_3) = 1, F(s_1) = 2, F(s_2) = 3, F(s_0) = 4\,.$$

Hence to adapt the SCC algorithm to featured transition systems, we construct a *tree* that symbolically represents all the possible finishing times of states.

Each path in such a *symbolic finishing-times tree* from the root to a leaf node represents a unique set of finishing times for the states in a featured transition system. The tree is annotated with feature-expression labels on edges, associating products with states' finishing times. Specifically, a tree node representing state s at level d in the tree means that the finishing time of state s is $|S|-d+1$ in *all products* that satisfy the feature expressions along the path from the root to the node.

For example, the WFTS from Fig. 2 gives rise to the symbolic finishing-times tree shown in Fig. 3. This tree assigns one set of finishing times for all products that contain either feature G or A, and another set of finishing times for products that contain neither feature.

Definition 1. *Let fts be a featured transition system. A* ***symbolic finishing-times tree*** *for fts is composed of a tree $T = (V, E)$ of height $n = |S|$, a node labelling function $\ell_v : (V \setminus root) \to S$ and a function $\ell_e : E \to \mathbb{B}(N)$ which labels each edge with a feature expression. The tree T satisfies the following conditions:*

- *All leaf nodes are at level $|S|$ of the tree.*
- *For any path $v_0, \ldots, v_n$ from the root to a leaf node, each node v_i is mapped to a unique state: $\forall i, j \in \{1 \ldots n\}, i \neq j : \ell_v(v_i) \neq \ell_v(v_j)$. A path from the root to a leaf node represents a set of products that share the same finishing times for its nodes.*
- *The feature expressions of outgoing edges from a node are disjoint: $\forall (u, v), (u, w) \in E, w \neq v : [\![\ell_e((u, v))]\!] \cap [\![\ell_e((u, w))]\!] = \emptyset$.*
- *For any product p and level i, there exists a (necessarily unique) path $v_0, \ldots, v_i$ from the root to a node in level i such that the product p is contained in the conjunction of the feature expressions along the edges of the path: $\forall p \in [\![d]\!], i \in \{1, \ldots, n\} : \exists$ a path $v_0, \ldots, v_i \ : p \in \bigcap_{j=0}^{i-1} [\![\ell_e((v_j, v_{j+1}))]\!]$.*

Alg. 1 Featured transition system depth first search

```
1   Procedure DFS-Fts (G)
2   begin
3       for each u ∈ V[G]
4           color[u][White] ← ⊤
5       time ← 0
6       for each u ∈ V[G]
7           if color[u][White] is satisfiable
8               DFS-Fts-Visit(u, color[u][White])
9           end-if
10  end
11  Procedure DFS-Fts-Visit(u, λ)
12  begin
13      Exploring ← color[u][White] ∧ λ
14      color[u][White] ← color[u][White] ∧¬λ
15      for each (u, v, λ′) ∈ E[G]
16          NextFExp ← λ′ ∧ λ
17          if (color[v][White] ∧ NextFexp) is sat.
18              DFS-Fts-Visit(v, NextFexp)
19          end-if
20      time ← time + 1
21      O[u][Exploring] ← time
22  end
```

- *For any product p, level i, and the unique path from the root $v_0, \ldots, v_i$ such that $p \in \bigcap_{j=0}^{i-1} [\![\ell_e((v_j, v_{j+1}))]\!]$, the finishing times in the projection $fts_{|p}$ of the states $\ell_v(v_1), \ldots, \ell_v(v_i)$ are $n, \ldots, n-i+1$, respectively.*

The symbolic finishing-times tree is built in two phases. In the first phase (performed by Alg. 1), a symbolic depth-first search explores all states of an FTS and computes a temporal ordering for when a state and all of its neighbors are explored, depending on feature expressions. The second phase (shown in appendix) uses this information to construct a symbolic finishing-times trees in a breadth-first manner.

In Alg. 1, unlike in a standard depth-first algorithm, states are not marked as visited by a boolean flag, but instead with a feature expression representing *under which set of products* they have been visited. Hence Alg. 1 stores and updates an array **White** of boolean formulas: representing the products for which a state has not been explored

Algorithm 1 starts by initializing array White to true (all products) for each state (lines 3-4). It then iterates over all states, and for each state that has not been fully explored, it calls the subroutine DFS-Fts-Visit with that state and the feature expression representing the set of unexplored products as parameters (lines 6-8).

The subroutine DFS-Fts-Visit starts by updating (reducing) the set of unexplored products for its given state (line 13). Then it iterates over each outgoing edge and checks if there are products for which the target state has not been explored, i.e. if color[v][White] $\wedge\lambda'\wedge\lambda$ is satisfiable (line 17). If so, then it recursively calls itself to explore the destination state. Finally, once all outgoing edges have been explored, it sets the finishing time for the given state and feature expression to the current time counter and increments this counter.

Once the feature-based depth-first ordering of states has been computed, this data can be used to construct the symbolic finishing-times tree for the FTS. We do this by iterating over the states in reverse finishing order, recursively adding a new child to a tree node whenever a new pair (s, λ) is found for which λ is not contained in the disjunction of the feature expressions along the edges to the other children. The algorithm, together with a precise explanation, can be found in appendix.

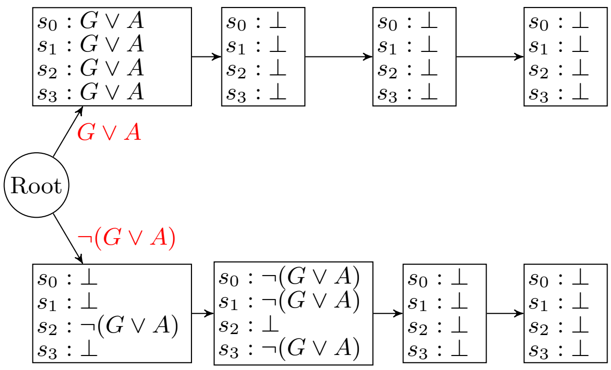

Figure 4: Symbolic SCC tree for the WFTS of Fig. 2

4.2 Strongly Connected Components of a Featured Transition System

After building the symbolic finishing-times tree, we use this tree to compute the SCCs of an FTS. We adapt the standard algorithm for computing SCCs (see Sect. 2.1) by replacing the single set of finishing times by the symbolic finishing-times tree. Hence we no longer compute a single set of SCCs, but instead compute one such set for each path from the root to a leaf node in the tree. This adaptation is necessary as the "finishing times" of states in an FTS depend on which features are present in a given product.

We explore each path from the root to a leaf node of the symbolic finishing-times tree. In the standard SCC algorithm, a boolean array keeps track of which states have been assigned to an SCC. In our case, we use an array of feature expressions representing *for which products* a state has been assigned. The algorithm to compute the symbolic SCCs is shown as Alg. 2. It uses a subroutine VisitDFS-For-SCC which we show in appendix.

The output of Alg. 2 is a *symbolic SCC tree*. Its tree structure is the same as the symbolic finishing-times tree, but now the tree nodes are labeled with mappings from S to $\mathbb{B}(N)$, representing for which products a given state is assigned to a particular SCC. As an example, the (very simple) symbolic SCC tree of the grant/request WFTS is displayed in Fig. 4.

Algorithm 2 starts by successively exploring each outgoing edge from the root of the tree (line 7). It then adds a triplet consisting of the child of the root node, along with its state and feature expression labels, to a stack of nodes to explore (lines 9-11).

The algorithm then enters a loop where elements of the stack are processed (lines 13-28), which corresponds to a depth first exploration of the finishing times tree. A triplet of tree node, state and feature expression is peeked from the stack (without being popped). The feature expression is compared to $R'(s)$ which contains the set of products for which the given state is already assigned to an SCC, and if it is not contained in $R'(s)$, then a new symbolic SCC is computed by calling VisitDFS-For-SCC (line 16-17). The

Alg. 2 Computing strongly connected components for an FTS given a symbolic finishing-times tree.

1 **Procedure** SymbolicSCC
2 **Input:** T, NodeLabel, EdgeLabel: a symbolic finishing-times tree
3 **Output:** RC: A function from tree nodes to symbolic SCCs
4 **begin**
5 NodesToExplore $\leftarrow$ empty stack of triplets of tree nodes, states and feature expressions
6 ReachabilityStack $\leftarrow$ empty stack of mappings $S \rightarrow \mathbb{B}(N)$
7 **For each** $e =$ (Root(T), u) $\in E(T)$
8 $R' \leftarrow \{\}$
9 $\lambda_0 \leftarrow$ EdgeLabel(e)
10 $s_0 \leftarrow$ NodeLabel(u)
11 NodesToExplore.push((u, s_0, λ_0))
12 ReachabilityStack.push(R')
13 **while** NodesToExplore $\neq$ [] **do**
14 u, s, $\lambda \leftarrow$ NodesToExplore.peek()
15 Visited(u)$\leftarrow$ True
16 **if** $\lambda \wedge \neg R'(s)$ is satisfiable
17 RC(u) $\leftarrow$ VisitDFS-For-SCC $(s, \lambda \wedge \neg R'(s), R')$
18 $R' \leftarrow R' \cup RC$(u)
19 **end-if**
20 Take v in Children(u) with Visited(v)=False
21 **if** no such v exists:
22 NodesToExplore.Pop();
23 $R' \leftarrow$ ReachabilityStack.Pop()
24 **else**
25 $\lambda' \leftarrow$ EdgeLabel(u,v)
26 NodesToExplore.push((v, NodeLabel(v), $\lambda \wedge \lambda'$))
27 ReachabilityStack.push(R')
28 **end-if**
29 **return** RC
30 **end**

Alg. 3 Computation of the maximum mean weight cycle in an SCC.

1 **Procedure** Mean-Cycle-SCC()
2 **Input:** $R : S \rightarrow \mathbb{B}(N)$: a symbolic SCC
3 **Output:** $C : \mathbb{B}(N) \rightarrow \mathbb{R}$: a symbolic maximum mean-weight cycle
4 **begin**
5 Pick $s_o \in S$: an arbitrary initial state
6 **for** k $= 0, \ldots, n$ and $v \in S \setminus \{s_0\}$
7 D[k, v, R(v)] $\leftarrow -\infty$
8 D[0, s_0, R(s_0)] $\leftarrow 0$
9 **for** k $= 1, \ldots, n$ and $v \in S$
10 **for** $(u, \alpha, v) \in \mathit{trans}$ s.t. $R(u) \neq \bot$
11 $\delta_1 = \gamma((u, v))$
12 **for** $\delta_2 \in$ domain(D[k, v, $\bullet$]) and $\delta_3 \in$ domain(D[k-1, u, $\bullet$])
13 **if** $\delta_1 \wedge \delta_2 \wedge \delta_3 \not\models \bot$ and D[k-1, u, δ_3] + W((u,α,v)) > D[k, v, δ_2]
14 D[k, v, $\delta_2 \wedge \delta_3 \wedge \delta_1$] $\leftarrow$ D[k-1, u, δ_3] + W((u,α,v))
15 D[k, v, $\delta_2 \wedge \neg(\delta_3 \wedge \delta_1)$] $\leftarrow$ D[k, v, δ_2]
16 Undef D[k, v, δ_2]
17 **end-if**
18 C[R(s_0)] $\leftarrow -\infty$
19 **for** $v \in S$
20 M[v, R(v)] $\leftarrow +\infty$
21 **for** k $= 0, \ldots, n-1$
22 **for** $\delta_1 \in$ Domain(M[v, $\bullet$]), $\delta_2 \in$ Domain(D[n, v, $\bullet$]), and $\delta_3 \in$ Domain(D[k, v, $\bullet$])
23 **if** $\delta_1 \wedge \delta_2 \wedge \delta_3 \not\models \bot$ and M[v, δ_1] > (D[n, v, δ_2] - D[k, v, δ_3])/(n-k)
24 M[v,$\delta_1 \wedge \delta_2 \wedge \delta_3$] $\leftarrow$ (D[n, v, δ_2] - D[k, v, δ_3])/(n-k)
25 M[v, $\delta_1 \wedge \neg(\delta_2 \wedge \delta_3)$] $\leftarrow$ M[v, δ_1]
26 Undef M[v, δ_1]
27 **end-if**
28 **for** $\delta_1 \in$ Domain(C[$\bullet$]) and $\delta_2 \in$ Domain(M[v, $\bullet$])
29 **if** $\delta_1 \wedge \delta_2 \not\models \bot \wedge$ C[δ_1] < M[v, δ_2]
30 C[$\delta_1 \wedge \delta_2$] $\leftarrow$ M[v, δ_2]
31 C[$\delta_1 \wedge \neg\delta_2$] $\leftarrow$ C[δ_1]
32 Undef C[δ_1]
33 **end-if**
34 **return** C

set of products assigned to an SCC for each state is then updated.

After processing the current tree node, the algorithm looks for a child that has not been explored (line 20). If no such child exists, then the current element is popped from the stack, otherwise a triplet is built from the child node, its state label and the feature expression labelling the edge to it and pushed to the stack of nodes to explore (lines 25-27). The algorithm continues processing triplets in the stack until it is empty and the complete finishing-times tree has been explored.

The procedure VisitDFS-For-SCC computes the set of states which are reachable from a given state s in the transpose of the input DFS, parameterized by feature expressions. This is inspired by the symbolic reachability algorithm of [7], except that here we exclude states from the search which have already been assigned to previous SCCs. The procedure is shown as Algorithm B in appendix.

4.3 Maximum Mean Cycle Computation

To complete the limit average computation, we need to identify the maximum mean cycle in a strongly connected component. We show the adapted algorithm as Alg. 3.

Our algorithm is a feature-aware variant of Karp's original algorithm [19]. As in Karp's algorithm, we chose an arbitrary initial state s_0 and start by computing a function D which for each state v and each $k \in \{0, \ldots, n\}$ gives the maximal weight of a path of length k from v to s_0. However, this weight will also depend on the feature guards along paths, so that D now takes a feature expression as extra input.

After initialization in lines 6-8, computation of D starts in line 9. For each pair k, v, $D[k, v]$ is defined on a *feature partition* of $R(v)$, the feature expression which governs whether v is present in the current SCC. Initially (line 7), the domain of $D[k, v]$ is the coarsest partition of $R(v)$, which is $R(v)$ itself, and during the iteration in lines 9-17, this partition is refined as necessary.

For each $k \in \{1, \ldots, n\}$, each $v \in S$, and each transition

(u, α, v), we need to check whether $D[k, v] < D[k-1, u] + W((u, \alpha, v))$, and if it is, update it to this value. Now both $D[k, v]$ and $D[k-1, u]$ are defined on (possibly different) feature partitions, and the transition (u, α, v) is only enabled for some feature guard δ_1. Hence we need to find each δ_2 in the domain of $D[k, v]$ and each δ_3 in the domain of $D[k-1, u]$ for which the conjunction $\delta_1 \wedge \delta_2 \wedge \delta_3$ is satisfiable (line 12) and then check whether $D[k, v, \delta_2] < D[k-1, u, \delta_3] + W((u, \alpha, v))$. If it is, then $D[k, v]$ needs to be updated, but only in the part of its partition where v can be reached from u, hence only at $\delta_1 \wedge \delta_2 \wedge \delta_3$. That is (lines 14-16), we need to split the domain of $D[k, v]$, update the value at $D[k, v, \delta_1 \wedge \delta_2 \wedge \delta_3]$, and keep the old value at $D[k, v, \delta_2 \wedge \neg(\delta_1 \wedge \delta_3)]$.

In the next part of the algorithm (lines 19-27), we compute $M[v] := \min_{k<n} \frac{D[n,v]-D[k,v]}{n-k}$ for each $v \in S$. As this again depends on the feature guards on the transitions, also $M[v]$ is defined on a feature partition which initially is set to $R(v)$ (line 20) and then refined as necessary. Finally, in lines 28-33, we use the same partition refinement technique once more to compute $C := \max_{v \in S} M[v]$, which per Karp's theorem [19] is the maximum mean cycle weight of the SCC.

5. IMPLEMENTATION AND EVALUATION

We have implemented our algorithms within ProVeLines, "a product line of verifiers for SPLs" [9]. ProVeLines takes as input specifications written in fPromela, a feature-aware extension of the Promela language [16], which we have extended to be able to specify transition weights. We have modified the code of ProVeLines (written in `C`) to include weights on transitions and perform a family-based and product-based computation of the maximum mean cycle. For our implementation, we have added 4300 lines of code to ProVeLines.

5.1 Subject Systems

For testing and experiments, we have implemented a variant of the taxi-shuttle example in which the number of extra licenses is parameterized. This variant has N different extra-license features $L_1, \dots, L_N$, each with their own Pickup-ext$_i$ and Release-ext$_i$ states and transitions a copy of the ones in Fig. 1, but guarded by the feature L_i. A formal description of this parameterized example is available in appendix.

We also tested the algorithm on an FTS representing a mine pump controller used in [6], with 2 optional features and 4 products. We annotated the transitions with artificial weights.

The taxi example had from 52 up to 2982 states, while the mine pump controller example had 9441 states.

5.2 Results

Table 1 shows the running times of our implementation, depending on the number of features $(N + 2)$, for both family-based and product-based analysis for the taxi example and the mine pump controller example. We ran both the family-based and product-based analysis ten times each. The family-based approach is faster than the product-based approach for the taxi example but not for the mine pump controller example.

5.3 Discussion

In the taxi example many products share the same symbolic strongly connected components. Hence the required time is reduced by using a family based-approach as a single computation over a symbolic strongly connected component can provide answers that can be re-used across multiple products.

We found that computing the maximum mean cycle for very large symbolic SCC was taking most of the time in the family based approach. Moreover the mine pump controller example has a much larger state space than the taxi example. Hence we decided to attempt to perform an abstraction of the mine-pump controller state space to improve performance.

The mine-pump controller has multiple processes running in parallel. It was not necessary to consider all possible interleavings of theses processes in order to consider all possible cycles. Hence we labelled some of its key states as important states and only considered transitions between them. We performed the computation over a much smaller state space and reduced the running times (to approximately 35 seconds and 6 seconds for the family-based and product-based approach respectively) for both approaches while still considering all cycles. However the product-based approach was still faster than the family-based approach for the mine pump controller example.

We also considered a different representation of strongly connected components for the family based approach. In this representation we used a binary tree with edges annotated with presence or not of a feature. Moreover each node would contain a set of all states that would be part of a SCC in any product satisfying the feature expression for the path from the tree to such node. Hence when computing the maximum mean-cycle we analyzed all the possible concrete SCCs in this tree. However this approach didn't improve the performance either as there was too little sharing of finishing times between products.

By annotating the code we have realized that different products induce different sets of finishing times over its states, and that there is very little sharing across products of symbolic strongly connected components. Therefore the family based approach doesn't improve the performance for this example and the overhead introduced by the family based approach means it is substantially slower than the product-based approach.

6. CONCLUSION AND FUTURE WORK

References

[1] S. Ben-David, B. Sterin, J. M. Atlee, and S. Beidu. Symbolic model checking of product-line requirements using sat-based methods. In *ICSE*, pages 189–199. IEEE Press, 2015.

[2] U. Boker, K. Chatterjee, T. A. Henzinger, and O. Kupferman. Temporal specifications with accumulative values. *ACM Trans. Comput. Log.*, 15(4):27:1–27:25, 2014.

[3] P. Černý, T. A. Henzinger, and A. Radhakrishna. Simulation distances. *Theor. Comput. Sci.*, 413(1):21–35, Jan. 2012.

[4] P. Černý, T. A. Henzinger, and A. Radhakrishna. Quantitative abstraction refinement. In *POPL*, pages 115–128. ACM, 2013.

Table 1: Average time consumption of Family-Based and Product-Based limit average computation on the taxi and the mine pump controller examples.

# of Features	# of Products	# of states	Family Based Time(s) (Mean ± Std. Dev.)	Product Based Time(s) (Mean ± Std. Dev.)
Taxi example				
3	8	52	0.25 ± 4.57 %	0.27 ± 9.44 %
4	16	75	0.30 ± 3.47 %	0.56 ± 1.64 %
5	32	98	0.44 ± 2.99 %	1.04 ± 9.04 %
6	64	121	0.80 ± 4.15 %	2.19 ± 2.19 %
7	128	144	1.83 ± 13.24 %	4.89 ± 1.36 %
8	256	167	3.86 ± 1.07 %	10.6 4 ± 2.01 %
9	512	190	10.84 ± 8.95 %	23.25 ± 2.10 %
10	1024	213	24.63 ± 6.26 %	51.71 ± 1.94 %
11	2048	236	63.27 ± 5.05 %	114.74 ± 1.79 %
12	4096	259	142.30 ± 5.27 %	251.87 ± 1.47 %
13	8192	282	307.56 ± 1.55 %	554.16 ± 1.33 %
Mine pump controller example				
2	4	9441	291.84 ± 2.79 %	110.91 ± 7.61 %

[5] P. Černý, M. Chmelík, T. A. Henzinger, and A. Radhakrishna. Interface simulation distances. *Theor. Comput. Sci.*, 560:348–363, 2014.

[6] 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 *ICSE*, pages 335–344. ACM, 2010.

[7] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Trans. Softw. Eng.*, 39(8):1069–1089, Aug. 2013.

[8] A. Classen, M. Cordy, P. Heymans, A. Legay, and P. Schobbens. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Sci. Comput. Program.*, 80:416–439, 2014. doi: 10.1016/j.scico.2013.09.019. URL `http://dx.doi.org/10.1016/j.scico.2013.09.019`.

[9] M. Cordy, A. Classen, P. Heymans, P. Schobbens, and A. Legay. ProVeLines: a product line of verifiers for software product lines. In *SPLC Workshops*, pages 141–146. ACM, 2013.

[10] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms.* McGraw-Hill, 2nd edition, 2001.

[11] U. Fahrenberg and A. Legay. General quantitative specification theories with modal transition systems. *Acta Inf.*, 51(5):261–295, 2014.

[12] U. Fahrenberg and A. Legay. The quantitative linear-time-branching-time spectrum. *Theor. Comput. Sci.*, 538:54–69, 2014.

[13] T. A. Henzinger. Quantitative reactive modeling and verification. *Computer Science - R&D*, 28(4):331–344, 2013.

[14] T. A. Henzinger and J. Otop. From model checking to model measuring. In *CONCUR*, volume 8052 of *LNCS*, pages 273–287. Springer, 2013.

[15] T. A. Henzinger and J. Sifakis. The discipline of embedded systems design. *IEEE Computer*, 40(10):32–40, 2007.

[16] G. J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004.

[17] R. Jain. *The art of computer systems performance analysis.* Wiley, 1991.

[18] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study. Technical report, Software Engineering Institute - CMU, 1990.

[19] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discr. Math.*, 23:309–311, 1978.

[20] K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. In *ASE*, pages 269–280. IEEE Computer Society, 2009.

[21] S. M. Shatz, J. Wang, and M. Goto. Task allocation for maximizing reliability of distributed computer systems. *IEEE Trans. Computers*, 41(9):1156–1168, 1992.

[22] T. Thum, S. Apel, C. Kastner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, June 2014.

[23] U. Zwick and M. Paterson. The complexity of mean payoff games on graphs. *Theor. Comput. Sci.*, 158:343 – 359, 1996.

Alg. A Algorithm to build a symbolic finishing-times tree for an FTS

1 **Procedure** ComputeTreeBfs(F, FInv)
2 **begin**
3 Q $\leftarrow$ Empty Queue
4 T $\leftarrow$ New Tree()
5 T.root.maxO $\leftarrow$ |domain(F)|
6 Q.add(T.root)
7 **while** ($\neg$Q.isEmpty())
8 Node $\leftarrow$ Q.pop()
9 $\lambda_1 \leftarrow$ FeatureExpressionFromRoot(Node)
10 max $\leftarrow$ Node.maxO
11 notChildren $\leftarrow \top$
12 j $\leftarrow$ max -1
13 **while** ($j > 0$)
14 u, $\lambda \leftarrow$ FInv(j)
15 **if** ($\lambda \wedge$ notChildren $\wedge\ \lambda_1$ is SAT)
16 NewNode $\leftarrow$ CreateNode(Node, u, $\lambda \wedge$ notChildren)
17 Q.add(NewNode)
18 notChildren $\leftarrow$ notChildren $\wedge \neg\lambda$
19 **end-if**
20 j $\leftarrow j - 1$
21 **return** T
22 **end**
23 **Procedure** CreateNode(ParentNode, State, λ)
24 **begin**
25 NewNode $\leftarrow$ New Node()
26 ParentNode.add(NewNode)
27 StateLabel(NewNode) $\leftarrow$ State
28 EdgeLabel(ParentNode, NewNode) $\leftarrow \lambda$
29 **end**
30 **Procedure** FeatureExpressionFromRoot(Node)
31 **begin**
32 **if** Node = T.root
33 **return** $\top$
34 **else**
35 **return** EdgeLabel(Parent(Node), Node) $\wedge$ FeatureExpressionFromRoot(Parent(Node))
36 **end-if**
37 **end**

APPENDIX

A. CONSTRUCTING A SYMBOLIC FINISHING-TIMES TREE

Algorithm A builds a symbolic finishing-times tree for an FTS in a breadth-first manner. It uses the order numbers generated by Alg. 1 for pairs of states and feature expressions stored in the injective partial function O, as well as an inverse function of it (denoted OInv) mapping an order number to a pair of state and feature expression.

The algorithm starts by initializing a tree T with an empty root node and adding it to a queue of tree nodes to explore (lines 3-6). It then enters a loop where it processes tree nodes from the queue and computes all their children (lines 7-20).

In order to identify all children of a tree node, the algorithm iterates over order numbers lower than than the maximum order number stored in the tree node in decreasing order (lines 13-20). It searches for pairs of states and feature expressions $(s, \lambda) = \text{OInv}(i)$ such that the feature expression (λ) combined with the negation of all other edges leaving the tree node is satisfiable (line 15-19). If the feature expression is satisfiable, then it adds the new children to the tree (line 16) and updates the expression representing the negation of all edges leaving the tree node (line 18). It records the order number in the tree node and then adds the new tree node to the queue (line 17). After all children for a tree node have been identified and added, any tree nodes remaining in the queue are processed (line 7).

Alg. B Reachability computation for the transpose of an FTS, excluding states already assigned to an SCC.

1 **Procedure** VisitDFS-For-SCC(s_0, λ_0, R')
2 **Inputs:** s_o: initial state of the SCC
λ_0: initial feature expression of the SCC
$R' : S \rightarrow \mathbb{B}(N)$: the (symbolic) set of states which are already assigned to an SCC, to exclude them
3 **Output:** $R : S \rightarrow \mathbb{B}(N)$
4 **begin**
5 $R \leftarrow \{(s_0, \lambda_0)\}$
6 Stack.push((s_0, λ_0))
7 **while** Stack $\neq$ [] **do**
8 (s, px) $\leftarrow$ Stack.peek()
9 new $\leftarrow \{(s', px') \in \text{Post}\ (s, px) \mid px' \not\subseteq R(s') \cup R'(s')\}$
10 **if** new $= \emptyset$ **then**
11 pop(Stack);
12 **else**
13 Take $(s', px') \in$ new
14 $R(s') \leftarrow R(s') \cup (px' \cap \neg R'(s))$
15 Stack.push($(s', px' \cap \neg R'(s))$)
16 **end-if**
17 **return** R
18 **end**

B. REACHABILITY FOR THE TRANSPOSE OF AN FTS

Algorithm B is a modified reachability search that takes as input an initial state, feature expression and symbolic set of excluded states, and computes the symbolic set of states reachable from the initial state and feature expression without going through any of the excluded states. It is similar to the symbolic reachability algorithm in [7], except we also keep track of a set of excluded states. This modified reachability algorithm returns a symbolic set of states: a mapping of states to feature expressions representing the set of states reachable under a given product.

Algorithm B starts by initializing an empty reachability relationship R with the initial state and feature expression and pushing the initial state and feature expression into a stack (line 7-9). It then enters a loop where it processes elements of the stack until the stack is empty (lines 10-20).

The algorithm peeks at the top element of the stack and computes the set of its successors that are not a member of either R or of excluded states R' (lines 11-12). If this set of new elements is empty then it pops the top element of stack (lines 14-15). Otherwise it takes a state and feature expression that is a new element, updates R with it and pushes the new element into the stack (lines 17-19). It then continues processing elements of the stack until no more remain and then returns R.