Quantitative Analyses of Software Product Lines

by

Rafael Ernesto Olaechea Velazco

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2022

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Dr. Juergen Dingel Professor, School of Computing, Queen's University
Supervisor(s):	Dr. Joanne Atlee Professor, School of Computer Science, University of Waterloo Dr. Krzysztof Czarnecki Professor, Department of Electrical and Computer Engineering, University of Waterloo
Internal Member:	Dr. David Toman Professor, School of Computer Science, University of Waterloo
Internal Member:	Dr. Richard Trefler Associate Professor, School of Computer Science, University of Waterloo
Internal-External Member:	Dr. Arie Gurfinkel Associate Professor, Department of Electrical and Computer Engineering, University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

A software product-line (SPL) is a family of related software systems that are jointly developed and reuse a set of shared assets. Each individual software system in an SPL is called a *software product* and includes a set of mandatory and optional *features*, which are independent units of functionality. Software-analysis techniques, such as model checking, analyze a model of a software system to determine whether the software system satisfies its requirements. Because many software-analysis techniques are computationally intensive, and the number of software products in an SPL grows exponentially with the number of features in an SPL, it tends to be very time consuming to individually analyze each product of an SPL. Family-based analyses have adapted standard software-analysis techniques (e.g., model checking, type checking) to simultaneously analyze all of the software products in an SPL, reusing partial analysis results between different software products to speed up the analysis. However, these family-based analyses verify only the functional requirements of an SPL, and we are interested in analyzing the quality of service that different software products in an SPL would exhibit. Quantitative analyses of a software system model (e.g., of a weighted transition system) can estimate how long a system will take to reach its goal, how much energy a system will consume, and so on. Quantitative analyses are known to be computationally intensive. In this thesis, we investigate whether executing a family-based quantitative analysis on a model of an SPL is faster than individually analyzing every software product of the SPL.

First, we present a family-based trace-checking analysis that facilitates the reconfiguration of a dynamic software product line (DSPL), which is a type of SPL in which features can be activated or deactivated at runtime. We assessed whether executing the family-based trace-checking analysis is faster than executing the trace-checking analysis on every software product in three case studies. Our results indicated that the family-based trace checking analysis, when combined with simple data-abstraction over an SPL model's quality-attribute values to facilitate sharing of partial-analysis results, is between 1.4 and 7.7 times faster than individually analyzing each software product. This suggests that abstraction over the quality-attribute values is key to make family-based trace-checking analysis efficient.

Second, we consider an SPL's maximum long-term average value of a quality attribute (e.g., because it represents the long-term rate of energy consumption of the system). Specifically, the maximum limit-average cost of a weighted transition represents an upper bound on the long-term average value of a quality attribute over an infinite execution of the system. Because computing the maximum limit-average cost of a software system is computationally intensive, we developed a family-based analysis that simultaneously computes

the maximum limit-average cost for each software product in an SPL. We assessed its performance compared to individually analyzing each software product in two case studies. Our results suggest that our family-based analysis will perform best in SPLs in which many products share the same set of strongly connected components.

Finally, because both of our family-based analyses require as input a timed (weighted) behaviour model of a Software Product Line, we present a method to learn such a timed (weighted) behaviour model. Specifically, the objective is to learn, for each transition t, a regression function that maps a software product to a real-valued weight that represents the duration of transition t's execution in that software product. We apply supervised learning techniques, linear regression and regularized linear regression, to learn such functions. We assessed the accuracy of the learnt models against ground truth in two different SPL and also compared the accuracy of our method against two different state-of-the-art methods: Perfume and a Performance-Influence model. Our results indicate that the accuracy of our learnt models ranged from a mean error of 3.8% to a mean error of 193.0%. Our learnt models were most accurate for those transitions whose execution times had low variance across repeated executions of the transition in the same software product, and in which there is a linear relationship between the transition's execution time and the presence of features in a software product.

Acknowledgements

I would like to thank Prof. Joanne Atlee for all her guidance during my graduate studies. She taught me how to do research, how to present my research, and how to choose appropriate research questions. I would also like to thank my other supervisor, Prof. Krzysztof Czarnecki, for his guidance and feedback.

I would like to thank my thesis committee: Prof. David Toman, Prof. Richard Trefler, and Prof. Arie Gurfinkel, for providing me with feedback that lead me to improve this thesis. I would also like to thank, Dr. Axel Legay and Dr. Uli Farhrenberg, for hosting me for a research visit in France that was very enjoyable and lead to the first paper that forms part of this thesis.

I would also like to thank my labmates, both from the GSD Lab and WatForm, for their feedback and for making my graduate school experience enjoyable. Finally, I would like to thank my family for all their support during my graduate studies.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Software product-line (SPL) *engineering* is a methodology for developing a family of related software-intensive systems while taking advantage of the commonalities between the individual systems [29]. A set of *features* – that is, units of functionality - are identified and created when developing an SPL. Features can be either mandatory or optional. A software *configuration* comprises all of the mandatory features and a subset of the optional features. Each individual system that is part of the SPL is denoted a *software product*. Each software product is associated with a corresponding configuration, which lists the set of features that the software product includes.

Standard software-analysis techniques, such as software model checking [4], analyze a software system (or a model of a software system) to determine whether it satisfies its requirements. As many software-analysis techniques are computationally expensive and the number of software products in an SPL grows exponentially with the number of features in an SPL, it is generally too time consuming to analyze individually each software product of an SPL. *Family-based analyses* adapt standard software-analysis techniques to simultaneously analyze all of the configurations of a software system, reusing partial analysis results across different configurations to speed up the analysis. For example, *software model checking* [4] is an automated decision procedure that analyzes a model of a software system, typically a state-transition model, to determine whether the system satisfies its functional requirements, typically expressed as temporal logic formulas [36]. To develop a family-based model-checking algorithm, Classen *et al.* [26] introduced a *featured transition system* (FTS), which represents, in a unified model, the state-transition models

of all the products of an SPL. They show that by analyzing the aforementioned FTS they can determine which products of an SPL satisfy a given temporal logic formula, and they also show, through a series of case studies, that such analysis is faster than separately model checking each individual SPL product. Family-based analyses have been developed for type checking [69], real-time model checking [31], SAT-based model checking [6], and probabilistic model-checking [105].

However, these family-based analysis techniques verify only the functional requirements of an SPL, whereas we are interested in analyzing as well the quality of service that the different software products in an SPL would exhibit. Quality attributes [49] represent how well a software system performs its functionality with respect to different facets of quality (e.g., how quickly the system performs its functionality, how much energy it consumes during execution, and how reliably it performs its functionality). For every relevant quality attribute, a system typically must satisfy some minimum quality requirement (specified as a constraint) and otherwise optimize performance on such quality attribute.

Many research projects address the problem of searching for software products that are optimal with respect to multiple, possibly conflicting, quality attributes [51, 54, 59, 95]. These tools predict the values of quality attributes for each software product based on a coarse-grained model. This coarse-grained model associates each feature with a static impact on each of the quality attributes. This model then simply aggregates (using simple arithmetic functions) each feature's impact to obtain a coarse estimate of each quality attribute for a configuration. Because a coarse-grained model accounts only for a feature's static contribution to a a quality attribute, these tools cannot account for fluctuations in a quality attribute due to changes in the execution environment or changes in the usage patterns of the system. For example, the energy consumption of an Unmanned Air Vehicle (UAV) depends not only on which features its configuration includes, but also on the actions executed by such system at runtime (e.g., activating the global positioning system sensors, taking photos, and so on).

An alternative approach is to augment a state-transition model of a system, which models the individual steps of that system's execution, with information about the impact of the execution of each transition on each quality aspect of interest. For example, in a state-transition model of the aforementioned Unmanned Air Vehicle, one transition could correspond to the UAV identifying its location based on GPS information and take 12 time units to execute, another transition could correspond to the UAV planning its path and take 28 time units to execute, and so on. To permit reasoning about the quality aspects of a system, *weighted transition system* models extend transition-system models by annotating their transitions with weights [4] that represent the effect that executing transitions have on quality aspects of interest. For example, a weight could represent the amount of energy

consumed by executing a transition or the time taken to execute a transition.

Quantitative analyses [20, 55] of these weighted transition systems permit estimating how much energy a system will consume [19], determining the how long will a system take to achieve its goal [15, 64], estimating by how much the system implementation deviates from its specification [16], and so on. It is generally believed that analysis of these weighted transition systems can provide a more accurate estimate of a software product's quality-attribute values than a static estimate based only on which features are present in the software product [50, 98]. However, quantitative analyses of these weighted transition systems can be computationally intensive.

Because family-based analyses of SPL's behaviour models have been shown to be faster than analyzing each software product individually [6, 26], we hypothesized that:

> *Hypothesis:* Executing a family-based quantitative analysis of an SPL will tend to be faster than separately executing the quantitative analysis for every software product of the SPL.

1.2 Thesis Overview and Contributions

In this thesis, we investigated this hypothesis by developing two family-based quantitative analyses of SPLs, a trace checking family-based analysis [87] and a worst-case long-term average family-based analysis [88], and by assessing their performance on case studies. Additionally, because manually creating these models is challenging, we explored how to extract a weighted behaviour model of an SPL from a set of execution traces of a sample of the software products in an SPL and we assessed accuracy of those extracted models.

> *Thesis Statement:* performing a family-based quantitative analysis on a model of an SPL can be faster than separately performing the quantitative analysis on every software product of the SPL, especially when combined with abstraction over the quality-attribute values.

The scope of applicability of this thesis is restricted to *annotative* software product lines. An annotative [102] software product line is a software product line that is implemented by annotating parts of artifacts (e.g., a transition model or source code) to indicate which products will include such part of the artifact. Thus, a software product line that is implemented via an FTS is an annotative software product line. Additionally, the scope

of applicability of this thesis is restricted to boolean SPLs. A boolean SPL is a software product line in which a feature can either be part or not be part of a configuration, and in which a configuration can have at most a single instance of a feature.

First, in Chapter 3, we present a family-based trace-checking analysis to facilitate the reconfiguration of *dynamic software product lines* (DSPLs) [60], which are a special type of SPL in which features can be enabled or disabled at runtime. As the execution environment of a DSPL changes, a different DSPL's configuration can become optimal with respect to quality of service. Because a DSPL has to make reconfiguration decisions at runtime, any analysis that is performed at runtime to determine which configuration would be ideal under the observed environment must execute quickly to be useful. *Trace checking* [43] is a lightweight quality-assurance technique that analyzes a quality of service that a (single) software system would provide over an observed execution trace. We assessed our hypothesis that a family-based trace-checking analysis would be an efficient improvement over analyzing all configurations of a DSPL. And found that the family-based analysis was not always faster because the differences in quality-of-service provided by many of the DSPL's configurations varied only slightly but each variation introduced a new diversion in the family-based analysis. To address this problem, we introduced a simple data-abstraction over the values of quality attributes to facilitate sharing of partial analysis results in our family-based approach. With the data abstraction, our family-based analysis was between 1.4 and 7.7 times faster than analyzing each configuration individually in all case studies.

In Chapter 4, we present a family-based analysis to compute the worst-case long-term average of a quality-attribute value for an SPL. The *limit-average cost* of a weighted execution trace is the average of the weights of the trace as the length of the trace goes to infinity. It can represent an upper bound on the long-term rate of energy consumed, or the degree of correctness of a software implementation. The standard algorithm [107] to compute the worst-case limit-average cost consists of a two-step process: first it computes the strongly connected components in a transition system, and then it identifies the highest mean-weight cycle in each strongly connected component. As we describe in detail in Chapter 4, we modified this algorithm to simultaneously analyze all of the software products of an SPL instead of separately analyzing each software product. We assessed the performance of our family-based algorithm compared to the cost of analyzing every product individually on two case studies. We obtained mixed results: our family-based approach is slightly faster than analyzing every product when the SPL has repetition in its behaviour and has a small number of states, whereas it is slower when the SPL with a large number of control states and a complex behaviour model.

Finally, in Chapter 5 we present a method to learn a timed (weighted) behaviour model of a software product line, which our analyses of quality attributes require as input. The

objective is to learn, for each transition, a weight function that maps software products to real-valued weights, so that the output of such function, when applied to an individual software product, is an accurate estimate of the transition's weight in that individual product. We apply supervised learning methods to learn, for each transition, a regression function that estimates how long the transition will take to execute in each software product that includes it. To assess the accuracy of transitions' execution-time estimates obtained by our proposed method, with respect to the observed execution-time of the transition in individual software products, we performed two case studies: a self-driving car and an open-source video encoder. Our results indicate that our approach produces WFTS models that can predict the time that each transition takes to execute in a software product with a mean error that ranges from 3.8% to 193.0%. We discuss the reasons for such large disparities in our estimate's accuracy, and identify an inverse relationship between the relative standard deviation of the execution times of a transition and the accuracy of our method in estimating its execution time. Despite the inaccuracies in the estimates for individual transitions, we are able to leverage our transition execution-time estimates and a count of the number of times each transition is executed to predict the overall execution time of an SPL product on an input task (e.g., encoding a video or compressing a file) with high accuracy, which is comparable to the accuracy obtained by a state-of-the-art method [98] that bases its estimates only on which features are present in a software product and on observing a sample of software products execute the input task (e.g., encoding a video or compressing a file). We also compare the accuracy of our per-transition execution-time estimates versus the accuracy of estimates obtained by Perfume [86], which is a state-of-the-art tool to extract timed behaviour models (i.e., a weighted transition system model) from timed execution traces.

1.3 Thesis Organization

The rest of this thesis consists of five chapters. In Chapter 2, we present background concepts that are used throughout the entire thesis. Chapter 3 presents a family-based trace-checking analysis for dynamic software product lines, Chapter 4 presents a family-based analysis for the worst-case long-term average value of a quality attribute of a software product line, and Chapter 5 presents a method to learn a timed behaviour model (WFTS) of a Software Product Line. In Chapter 6, we present our conclusions and future work. We discuss the related work in each chapter of the thesis.

Chapter 2

Background

In this chapter we introduce notation and background concepts that are used throughout the thesis. In Section 2.1 we present a summary of software product-line (SPL) concepts and models. In Section 2.2 we present an overview of formalisms to model the behaviour of software systems. In Section 2.3 we review dynamic software product lines (DSPLs). Finnally, in Section 2.4 we summarize a modelling language and tool that permit representing and analyzing software product line models.

2.1 Software Product Lines

Software product-line (SPL) engineering is a methodology to efficiently develop a collection of similar software systems from a set of core assets [29]. In SPL engineering, reuse of assets is planned and managed to take advantage of the commonalities that different software systems in the same domain share. When developing a set of related software systems, following an SPL engineering methodology can be beneficial because the software systems can be developed, tested, and maintained faster and more cheaply [94].

The development of a software product line is divided into the activities of *domain engineering* and *application engineering*. The goal of domain engineering is to analyze the problems for which the SPL is targeted, and to then identify and create a set of common assets that will be used across all or most of the systems in the SPL. The outcomes of domain engineering are: a variability model, a set of common requirements, a common platform, implementation components, and a test suite. In contrast, the goal of application engineering is to create a specific software system while re-using the existing platform

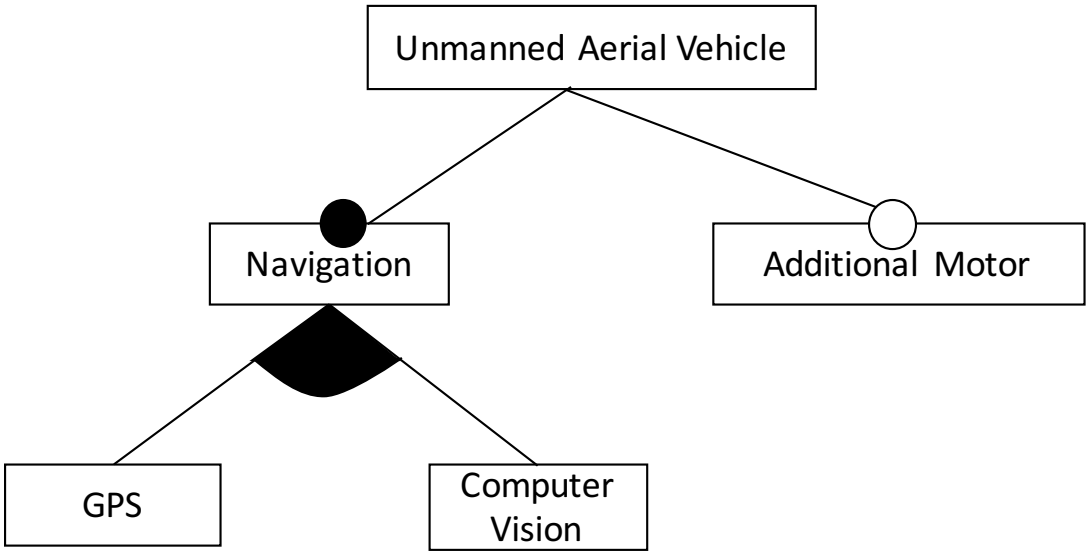


Figure 2.1: Feature model for an Unmanned Aerial Vehicle (UAV) Software Product Line.

and assets. In application engineering, the common assets are combined with each other and with product-specific assets to create an individual software system. Each individual software system is called a *software product*.

A software product line includes a set of *features*, which is a coherent unit of user-visible functionality. In the context of SPLs, a *feature* is a coherent unit of user-visible functionality. A feature can be either mandatory or optional. A *software configuration* comprises of mandatory features and a selected subset of optional features. The set of features that are included in a specific software product comprise that product's software configuration. We show the definition of a software product line below:

Definition 1 *A **software product line** comprises a set of boolean features that can be mandatory or optional. Let N be the set of all features in a software product line. A **software configuration** is a subset of features $px \subseteq N$ and the resulting software system composed of those features (features in px) is called a **product**. The **family of products** represented by the SPL is the set of products defined by allowable configurations $\subseteq 2^N$.*

However, not all combinations of features are valid software configurations. A *feature model* [66] is used to distinguish between valid and invalid software configurations. A feature model is visually represented as a hierarchical tree-like diagram whose nodes represent the features of an SPL and whose edges represent dependencies (or constraints) between

features. Figure 2.1 shows a feature model for an example Unmanned Aerial Vehicle (UAV) SPL. The Unmanned Aerial Vehicle SPL has four features: Additional Motor (M), Navigation (N), Global Positioning System (GPS), and Computer Vision (CV). A white circle indicates that a feature is optional (e.g., the Additional Motor feature) and a black circle indicates that a feature is mandatory (e.g,. the Navigation feature). The black semi-circle below the node for feature Navigation indicates that at least one of its child features, Global Positioning System or Computer Vision, must be selected in any configuration that includes feature Navigation. We show the definition of a feature model below:

Definition 2 *A **feature model** is represented as $d = (N, PX)$ where N is the set of all features and $PX \subseteq \mathcal{P}(N)$ is a subset of the power set of N that represents all valid configurations.*

Analysis of software product lines can be categorized into family-based or product-based [102]. *Product-based analysis* 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.

2.2 Behaviour Models

Transition systems are traditionally used to model and analyze the behaviour of software systems. A *transition system* (TS) contains a set of states of computation and transitions that capture allowable progressions of execution from one state of computation to another in response to the occurrence of actions. A set of initial states represent the possible starting states of a system's execution. More formally:

Definition 3 *A **transition system (TS)** is a tuple $ts = (S, I, A, T)$, where S is a set of states, $I \subseteq S$ is a set of initial states, A is a set of actions, and $T \subseteq S \times A \times S$ is a set of transitions.*

An *execution* π of a transition system is an alternating sequence of states and actions, $\pi = s_0\alpha_1 s_1\alpha_2 \ldots$, such that $s_0 \in I$ and for all execution steps $(s_i, \alpha_{i+1}, s_{i+1}) \in \pi$ corresponds to some transition in T. Additionally, each action can be labelled as either a system action (!) or an environment action (?).

A transition system represents the behavior of a single software product, whereas a software product line has many software product variants. To compactly model the behaviour

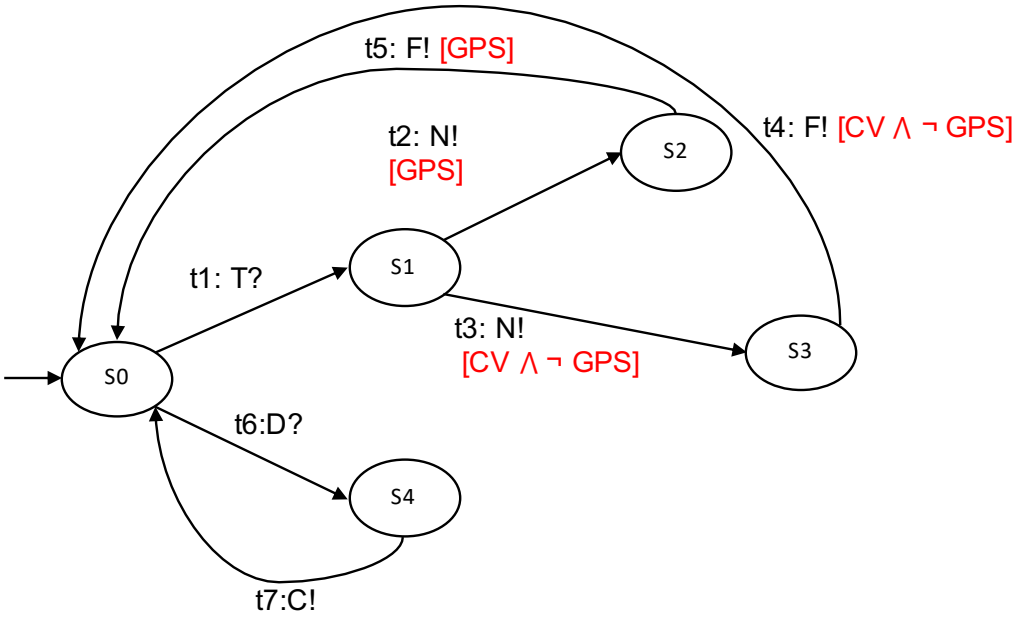


Figure 2.2: A featured transition system for the Unmanned Aerial Vehicle SPL. Feature expressions corresponding to each transition are displayed in red. Transitions that are not labelled with a feature expression are included in all products.

of all the products of an SPL in a single model, Classen *et al.* [26] introduce *featured transition systems* (FTS). Each optional feature is represented by a boolean *feature variable*, whose value denotes whether the feature is present (true) or not (false) in a product's configuration. Thus, a software configuration can be represented by an assignment of values to all the feature variables. Each transition is either i) annotated with a *feature expression* — a propositional formula ranging over feature variables — that denotes the set of products that exhibit that transition, or ii) is not annotated with a feature expression to denote that every product includes that transition. For example, the feature expression $GPS \wedge CV$ denotes all the configurations that include both feature Global Positioning System (GPS) and feature Computer Vision (CV). More formally:

Definition 4 *A **featured transition system (FTS)** is a tuple $fts = (S, I, A, T, d, \gamma)$, where (S, I, A, T) is a transition system, $d = (N, PX)$ is a feature model, and $\gamma : T \rightarrow \mathcal{P}(N)$ labels each transition with a feature expression.*

FTSs unify the transition systems of all the products of an SPL into a single annotated transition system. The FTS provides "a 150% model" of all the states and transitions in

an SPL's products – that is, it includes more transitions and states than are required for any of the SPL's individual products.

Software product lines that are implemented by annotating parts of an artifact (e.g., the transition model) with feature expressions to indicate which products will include such part (e.g., a specific transition) of the artifact are called *annotation-based* software product lines [102]. Thus, an SPL that is implemented through an FTS is an annotation-based software product line.

Figure 2.2 shows an FTS for a simple Unmanned Air Vehicle (UAV) product line. The UAV is assigned missions to either find and follow a target or to deliver a package. Environment action $T?$ models a mission to find and follow a target, whereas environment action $D?$ models a mission to deliver a package. Each transition is labeled with a feature expression, which is displayed in red, that indicates in which configurations that transition is present. For example, transition t3 is labelled with feature expression $CV \wedge \neg GPS$ to indicate that transition t3 will be present only in configurations that include feature Computer Vision and that do not include feature Global Positioning System. A transition system for a specific software configuration can be derived from the FTS model by including only the transitions whose feature expressions are satisfied by the configuration's feature-variable assignment. For example, the transition system for a configuration that includes only the feature GPS would exhibit only transitions t1, t2, t5, t6, and t7. The operation of deriving a transition system for a specific software configuration from a featured transition system is called "projecting" the featured transition system onto the software configuration. More formally:

Definition 5 *The **projection** of a Featured Transition System* $fts = (S, I, A, T, d, \gamma)$ *onto a software configuration* p *is given by the transition system* $ts = (S, I, A, T_p)$ *where* $T_p = \{t \in T |\ p \models \gamma(t)\}$.

To model the quality aspects of a system, transition systems (and featured transition systems) are extended with *weights* [4]. Transitions are annotated with a weight (also called a *reward* in the context of probabilistic model checking) that represents the effect of executing a transition on a quality aspect of interest. For example, the weight could represent the amount of energy consumed by executing a transition. The sum of weights in an execution trace then represents the total amount of energy consumed during that execution trace. A *weighted transition system* (WTS) models the quality of service (for a specific quality) on all transitions in a single configuration of an SPL, whereas a *weighted featured transition system* (WFTS) models the quality of service on all transitions in all configurations of an SPL. More formally:

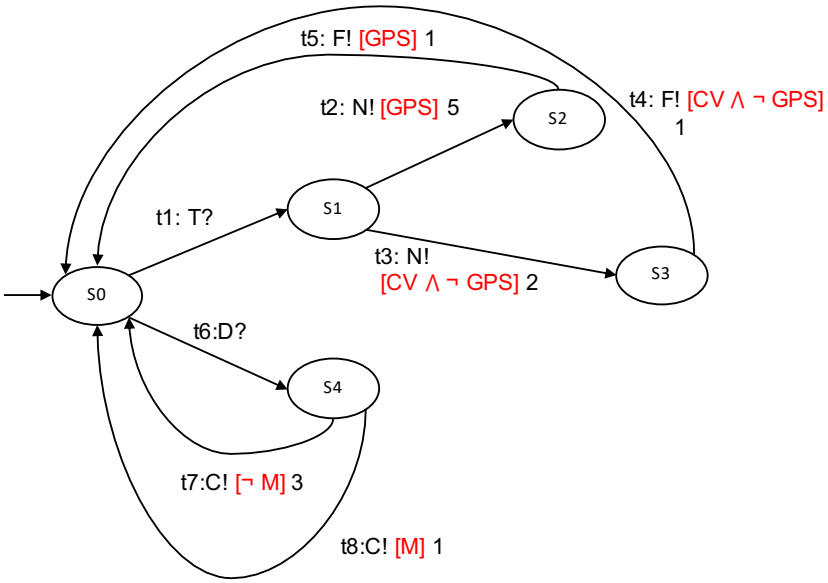


Figure 2.3: A weighted featured transition system for the Unmanned Aerial Vehicle SPL. Feature expressions corresponding to each transition are displayed in red and weights in black.

Definition 6 *A **Weighted Featured Transition System (WFTS)** is a tuple* $wfts = (S, I, A, T, d, \gamma, W)$*, where* (S, I, A, T, d, γ) *is an FTS and* $W : T \to (\mathcal{P}(N) \to \mathbb{R})$ *is a function that annotates each transition with a mapping from software products to weight values.*

Figure 2.3 shows a weighted featured transition system for the Unmanned Air Vehicle SPL. Each transition is annotated with a number that represents the cost of executing it.

2.3 Dynamic Software Product Lines

Dynamic software product lines (DSPL) have been developed [60] to permit re-configuring an SPL at runtime by enabling or disabling features on the fly, so that an SPL product can adapt to a changing environment, to changing user goals, and to failed components. Andersson and Bosch [3] show, through a series of industrial case studies, that there are companies using traditional SPL engineering to perform SPL runtime reconfiguration to satisfy changing quality requirements or hardware failures (or updates). DSPL research

leverages SPL engineering tools, processes, and concepts to facilitate building and testing systems that can adapt at runtime. In traditional software product-line engineering, variability is resolved at design time by selecting which of an SPLs optional features to include in the product being derived. In contrast, in a dynamic software product line, variability can be resolved and can be changed at runtime by dynamically activating or deactivating features.

Lee and Kang [74] introduce a feature-based approach for building a software system that can reconfigure itself at runtime (e.g., in essence a DSPL). They propose to group features into binding units, where each binding unit is assigned a time (either design-time, installation, or runtime) for when it can be activated (that is, when it will be decided whether to make its functionality active).

We show the definition of a Dynamic Software Product Line below:

Definition 7 *A **Dynamic Software Product Line (DSPL)** is a special type of Software Product Line that is extended with an **active configuration**. An active configuration represents the software product that is currently executing. The active configuration can be changed at runtime by activating or deactivating individual features at runtime, resulting in a change in the product that is executing from the product corresponding to the old active configuration to the product corresponding to the new active configuration.*

2.4 Promela, Featured Promela, and ProVeLines

In this subsection we give a brief overview of *Promela*, that is a modelling language that supports the modelling of software systems to enable model checking on them. We also summarize *Feature Promela*, which is an extension of *Promela* to represent behavioural models of software product lines. Finally, we provide an overview of *ProVeLines*, which is a family-based model checker for software product line models.

2.4.1 Promela

Promela [61] is a modeling language that was created to model distributed systems so that their correctness can then be verified using a model checker. Promela is the input language for the model checker SPIN [62].

A Promela program consists of processes, message channels, and variables. A process type is declared by using the keyword *proctype* followed by the name of the process type,

```
1 typedef features {
2   bool FeatureA;
3   bool FeatureB;
4 };
5 features f;
6 int i = 0;
7 proctype simpleproc() {
8   do ::
9       if  ::  f.FeatureA && f.FeatureB;
10              i = i + 3;
11          ::  else;
12              i = i + 2;
13      fi;
14  od;
15 }
```

Figure 2.4: The fPromela specification of an example SPL that has two features. In the product that includes both Feature A and Feature B, variable i is incremented by three on each iteration of the loop, whereas in the rest of the products variable i is incremented by two on each iteration of the loop.

a list of arguments enclosed in parenthesis, followed by the body of the process enclosed in curly brackets. The body of the process consists of a sequence of statements and declarations. The available statements are those generally provided by imperative programming languages (e.g., if statements, arithmetic operations, goto statements, assignments, and so on). The variables declared in the body of the process definition have "process scope" whereas variables declared outside a process definition have a global scope. Message channels can be used to communicate between processes and to synchronize the execution of different processes. Promela programs can include the instantiation of multiple processes. The execution of a Promela program is the parallel execution of each one of the instantiated processes. A full description of Promela is given in [62]. We note that a transition system can be represented by a Promela program.

2.4.2 Featured Promela

Featured Promela (*fPromela*) [25] extends Promela with "feature expressions" to permit representing featured transition systems. In an fPromela program, a structure denoted

"features" (a "C-style" struct) must be declared and each feature must be declared as a boolean variable inside that structure. Additionally, "if statements" in an fPromela program can be used to indicate that certain statements are executable only in products that satisfy certain feature expressions. More concretely, an "if statement" in an fPromela program can be either: i) a standard Promela "if statement" whose condition clauses do not refer to the "features structure" at all, or ii) a special "if statement" whose condition clauses comprise only feature expressions and the "else" keyword.

Figure 2.4 shows the fPromela code of a very simple example SPL that has two features: feature A and feature B. First, the program includes a declaration of a structure denoted "features" that includes two feature variables – one for Feature A and one for Feature B (lines 1-5). Second, a global variable i is created and initialized to zero (line 6). Third, a process type of name "simpleproc" that consists of a loop that executes an "if statement" is declared (lines 7-15). Finally, the first condition of the "if statement" is composed of the feature expression "f.FeatureA && f.FeatureB" to indicate that only products that satisfy the respective feature expression can execute that branch (lines 9-10). Thus, only products that satisfy "f.FeatureA && f.FeatureB" will execute the statement "i = i + 3" (line 10). The rest of the products would execute the statement "i = i + 2" that is part of the "else" branch of the "if statement" (lines 11-12). A full description of fPromela is given in [25]. We note that Classen *et al.* [25] prove that any featured transition system can be represented as an fPromela program and vice versa.

2.4.3 ProVeLines

ProVeLines [30] is a tool that performs family-based model checking of software product line models. It takes as input the behaviour model of a software product line represented as an *fPromela* program, and a feature model associated with that software product line that is represented either in *Textual Variability Language* (TVL) or as a boolean formula in conjunctive normal form (CNF). If the feature model is given as a TVL file, ProVeLines converts it into a boolean formula in conjunctive normal form before performing family-based model checking. ProVeLines can then verify whether all products of the given SPL satisfy a Linear Temporal Logic (LTL) formula or not. ProVeLines also provides facilities to parse fPromela programs and build your own family-based analysis on top of it. ProVeLines provides two options to represent feature expressions: *Ordered Binary Decision Diagrams* (BDDs) [12] or as boolean formulas in Conjunctive Normal Form. In this thesis, we use Ordered Binary Decision Diagrams to represent feature expressions. ProVeLines uses the Colorado University Decision Diagram (CUDD) [100] library to implement and manipulate BDDs. A more thorough description of ProVeLines is given in [25] and [30].

Chapter 3

Trace Checking for Dynamic Software Product Lines

An important objective of a dynamic software product line (DSPL) is to continually provide optimal quality of service while operating in a changing environment and changing usage patterns. As we discussed in Chapter 2, a DSPL has a set of optional features — units of functionality — that can be activated or deactivated at runtime; and a large but fixed number of configurations, which are defined by the set of active optional features. As the environment in which a DSPL is executing changes, the configuration can likewise change, in order to maintain optimal quality of service. A DSPL would benefit from knowing how each of its configurations would have performed (with respect to quality-of-service) in the recent past, as input to the decision of whether it should reconfigure and (if so) what the target configuration should be.

Dynamic software product lines need to adapt, at runtime, to a changing environment, changing user goals, or their own internal failures. To decide whether to reconfigure a system, existing approaches periodically analyze, at runtime, using computationally intensive techniques such as probabilistic model checking, the expected quality of service that each configuration would provide [13, 80]. Because these analyses are executed at runtime and analyze a large number of configurations, the efficiency of these analyses is a major concern [42, 44, 81].

We are interested in improving the efficiency of analyzing all configurations by using trace checking to analyze the performance (with respect to quality-of-service) of configurations as applied to an observed execution trace. *Trace checking* is a lightweight quality-assurance technique that analyzes a single execution of a software system to determine

whether the system satisfies its requirements on the observed execution. Trace checking has been applied to estimate the quality of service provided by a single software system on an observed execution trace [43]. Because the number of configurations of a DSPL increases exponentially with the number of optional features, performing a trace-checking analysis on every configuration of a DSPL can be time consuming.

In this chapter, we propose a *family-based analysis* that analyzes all the configurations simultaneously to speed up the analysis. We note that our family-based analysis does not have better worst-case complexity than product-based trace analysis, but instead we obtain a speed up based on potential sharing of partial-analysis results across different software products. Different configurations react to environment input, which is modelled by environment actions as indicated in Chapter 2, in different ways: transitioning to different states of computation and exhibiting different qualities of service. Our family-based trace-checking algorithm needs to track how the execution of different groups of configurations would have progressed if they had been executing, that is to which state(s) they would have transitioned and what quality-of-service they would have provided. As the execution of many software configurations (over a given trace) can have common behaviours, our algorithm groups together sets of software configurations that would result in the same system state and the same accumulated reward.

We evaluated the efficiency of our family-based analysis on three case studies taken from the literature. Our basic family-based analysis of the DSPL is between 1.1 and 5.7 times faster than analyzing each configuration individually in two out of the three case studies. The quality-of-service performance of many configurations vary only slightly between each other, which negatively impacts the ability of our family-based analysis to group together different configurations and reuse partial-analysis results across different configurations. Thus, we introduce a simple data abstraction over the values of quality attributes to facilitate sharing of partial-analysis results across different configurations, by clustering similar quality-of-service values together. With the data abstraction, our family-based analysis is between 1.4 and 7.7 times faster than the sum of analyzing each configuration individually in all case studies.

The main contributions of this chapter are:

- A family-based trace-checking algorithm (and its implementation) that analyzes the quality of service that every configuration of a DSPL would have provided over the recent system input.
- An evaluation of the empirical efficiency of such analysis on three DSPL case studies taken from the literature.

- Improvement over our initial analysis by applying a simple data abstraction over the values of quality attributes, which improves the efficiency of our family-based trace checking algorithm.

The rest of the chapter is organized as follows. In Section 3.1 we introduce an example DSPL that is used as a running example throughout the rest of the chapter, in Section 3.2 we review trace checking, and in Section 3.3 we present our family-based trace checking algorithm. In Section 3.4 we present the results of our evaluations and discuss our findings, and in Section 3.5 we present a data abstraction on the transitions' quality-attribute values and discuss how it impacts the efficiency of our analysis. In Section 3.6 we describe related work, and in Section 3.7 we present our conclusions.

3.1 Running Example

In Chapter 1 and Chapter 2 we introduced a running example that consists of an Unmanned Aerial Vehicle (UAV) that has to satisfy a series of mission requests that are either searching for a target or delivering a package to a specified location. We will use such example to illustrate how the active configuration of a DSPL can affect its runtime quality-of-service. This UAV has three optional features that can be activated or deactivated at runtime to help it complete its missions: a Global Positioning System (GPS) that can help it determine its location, a Computer Vision (CV) subsystem that can aid it with navigation, and an additional Motor (M) that can be activated to provide additional power.

When the UAV receives a mission to search for a target, the UAV can navigate towards the target either by relying exclusively on the computer vision subsystem or by relying on a combination of the GPS and the computer vision subsystem. A key objective for the UAV is to minimize the rate of energy consumption while still successfully completing its assigned missions. The GPS consumes energy at a high rate, so the UAV should activate the GPS only when it is necessary to locate its target. If the environment visibility is good, the computer vision subsystem will suffice to successfully guide the UAV to its target. If the UAV encounters an environment with low visibility, the UAV will require combining information from both the GPS and the computer vision subsystem to successfully reach its target.

Additionally, the UAV can be requested to deliver a package to a specified location. If the package is heavy, then the UAV may need to engage the extra motor, but at the cost of consuming more energy. If the package is light, the UAV would consume less energy by

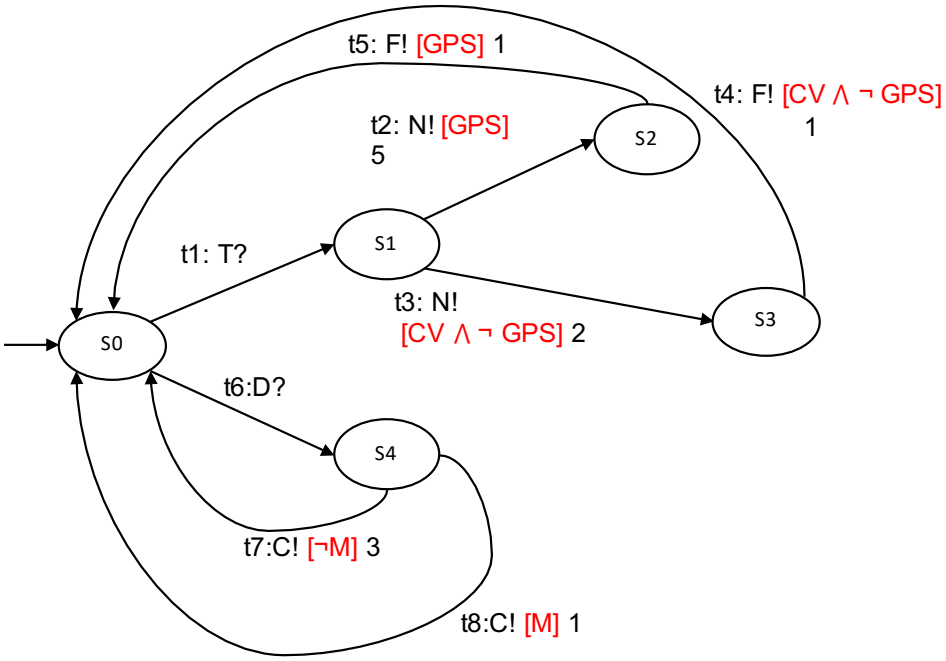


Figure 3.1: A weighted featured transition system for the unmanned aerial vehicle DSPL.

keeping the additional motor idle. Thus the ideal configuration of the DSPL varies with the conditions of the environment.

Figure 3.1 shows the weighted featured transition system for our running example of the Unmanned Air Vehicle (UAV) DSPL (it is exactly the same as the Figure 2.3 from Chapter 2, but we repeat it here for the convenience of the reader). As we stated in Chapter 2, the different mission assignments are modeled as environment actions: $T?$ for a mission to search for a target and $D?$ for a mission to deliver an object to a specific location. The responses of the UAV are modeled as system actions: $N!$ for navigating to its target, $C!$ for carrying an object to its destination, and $F!$ for following its target. The weights represent the units of energy consumed by the execution of each transition.

3.2 Background

In this section we review trace checking. An overview of transition systems, featured transition systems, weighted featured transition systems, and dynamic software product lines is already given in Section 2.2 and Section 2.3 of Chapter 2.

3.2.1 Trace Checking

Trace checking [40, 43] is a lightweight formal-analysis method that analyzes whether an observed execution trace of a system satisfies the system's functional or quality-of-service requirements. Trace checking can compute summary statistics, such as sum or average, about the weights associated with an execution trace of a weighted transition system [43]. These summary statistics can represent the quality of service provided by a configuration of a DSPL (represented by a weighted transition system) over an observed execution trace. For example, the sum of the weights over an execution trace of a system can represent the total amount of energy consumed during the execution of such system. We are interested in comparing the summary statistics of a DSPL's current configuration against the summary statistics of the DSPL's other configurations, in order to determine if the DSPL would perform with a better quality-of-service in another configuration. An observed execution trace can be simulated over different configurations of a DSPL (e.g., different weighted transition systems) to estimate the quality of service that each configuration would have provided. However, as the number of configurations of a DSPL grows exponentially with the number of features of a DSPL, this approach might be too time consuming when the number of features of a DSPL is large because it would require trace checking up to 2^N products where N is the number of features of the DSPL.

3.3 Approach

In this section we present a family-based algorithm that estimates the quality of service that each configuration of a DSPL would have provided over an observed execution trace. Subsequently, we extend this approach with data abstraction to improve its runtime performance.

3.3.1 Family-based Trace Checking

Different configurations react to environment inputs, which are represented as environment actions as stated in Chapter 2, in different ways — transitioning to different states of computation and exhibiting different qualities of service. Thus a family-based trace-checking algorithm needs to track how each configuration's execution would have progressed with respect to its sequence of states as well as its performance (quality of service). Our algorithm maintains a custom data structure that tracks for each software configuration the accumulated reward and system state that would result from the execution of that software

Alg. 1: Family-based simulation of a DSPL

Procedure Simulate-FTS-Execution(Trace)
Input: Trace $= s_0\alpha_0 s_1, s_1\alpha_1 s_2 \ldots$
Output: L: A set of tuples of state, feature expression, and accumulated reward.
begin
  L $\leftarrow \{(s_0, \top, 0)\}$
  for each $st = s_i, \alpha_i, s_{i+1} \in$ Trace
    $L' \leftarrow \{\}$
    if IsEnvironmentAction (α_i) **then**
      for each $(s, \gamma, r) \in$ L, $t = (s, \beta, s_{dst}) \in$ T, $\alpha_i = \beta$
        MergeTriplets(L' , $(s_{dst}, \gamma \wedge \gamma(t), r + W(t))$)
    else
      for each $(s, \gamma, r) \in$ L, $t = (s, \beta, s_{dst}) \in$ T s.t. $\gamma \wedge \gamma(t) \not\models \bot$ and IsSystemAction(β)
        MergeTriplets(L' , $(s_{dst}, \gamma \wedge \gamma(t), r + W(t))$)
    end-if
    L $\leftarrow L'$
  return L
end
Procedure MergeTriplets (M, (s, γ, r))
begin
    if $\exists\psi$ s.t. $(s, \psi, r) \in$ M **then**
      M $\leftarrow$ M $\setminus \{(s, \psi, r)\}$
      M $\leftarrow$ M $\cup \{(s, \gamma \vee \psi, r)\}$
    else
      M $\leftarrow$ M $\cup \{(s, \gamma, r)\}$
    end-if
end

configuration in response to a sequence of environment inputs: as the execution of many software configurations (over a given trace) can have common behaviours, our algorithm groups together sets of software configurations that would result in the same system state and same accumulated reward.

The custom data structure L records triplets of feature expressions, system state, and accumulated reward. The algorithm updates the data structure as actions from the exe-

cution trace are processed. It refines (splits) a tuple when different configurations in its feature expression would result in different accumulated rewards or system state. For example, a tuple $< \top, s_0, 0 >$ with feature expression $\top$ (representing all configurations), system state s_0, and an accumulated reward of 0 would be split into two different tuples after processing action $N!$: tuple $< GPS, s_2, 5 >$ resulting from the execution of transition $t2$, which has a weight of 5; and tuple $< CV \wedge \neg GPS, s_3, 2 >$ resulting from the execution of transition $t3$, which has a weight of 2. Similarly, the algorithm merges tuples if they share the same accumulated reward and resulting system state after processing an action; the feature expression in the merged tuple is the disjunction of feature expressions from the tuples being merged.

Our algorithm is listed in Algorithm 1. It starts by initializing the custom data structure L to contain a single triplet consisting of the feature expression $\top$ (representing all configurations)[1], the initial state, and an accumulated reward of zero (line 4). The algorithm then sequentially considers each action in the observed trace and determines the transitions that it triggers in all configurations.

When an environment action α is encountered (line 7), the algorithm determines the next state of execution of each configuration as a result of executing the transitions triggered by the configurations' respective current states and the action α. For each tuple $< s, \gamma, r >$ in L and for each transition t triggered by action α (line 8), a new tuple is generated that applies the effects of t. The generated triplet $< s_{dst}, \gamma \wedge \gamma(t), r + W(t) >$, has resulting state s_{dst} (the destination state of transition t); feature expression $\gamma \wedge \gamma(t)$ (the conjunction of the tuple's feature expression γ and t's feature expression $\gamma(t)$, which corresponds to the set of products that would reach state s_{dst}); and accumulated reward $r + W(t)$. For example, if L comprises two tuples $< s_o, GPS, 10 >$ and $< s_o, \neg GPS, 6 >$ when environment action D? occurs (which would trigger transition t6), then two new tuples would be generated: $< s_4, GPS, 10 >$ and $< s_4, \neg GPS, 6 >$ as transition t6 has destination state s_4 and a weight of 0.

As we saw in lines 1-9, when the current configuration reacts to an *environment action* α, the algorithm determines how other configurations react to the same event. In contrast, when the current configuration executes a *system action*, other configurations may execute different system actions (lines 11-14). Specifically, when the current configuration executes a system action α (line 10), our algorithm, for each tuple $< s, \gamma, r >$ in L (line 11), considers any transition t that has a matching source state s and whose feature expression $\gamma(t)$ is

[1]The algorithm uses feature expressions to represent sets of configurations. Our implementation of the algorithm uses *Ordered Binary Decision Diagrams* (BDDs) [12] to encode feature expressions. More concretely, our implementation of the algorithm uses the *Colorado University Decision Diagram* (CUDD) [100] library to implement and manipulate BDDs that represent feature expressions.

compatible with the tuple's feature expression (γ) (lines 11-12), to generate a new tuple (line 12).

After processing an action, the resulting triplets are collected into a new version of L, denoted L' . The procedure *MergeTriplets* is responsible for updating *L'* as new tuples are generated. *MergeTriplets* verifies whether a given pair of system state s and accumulated reward r already exists in M. If it does, then it updates the existing feature expression associated with them in L (ψ) to a new feature expression that is the disjunction of ψ and the new feature expression γ (lines 19-20). If the pair s and r doesn't already exist in some tuple in *L'*, then the algorithm updates *L'* to include the new tuple (s, γ, r) (line 22). We note that our algorithm uses *Ordered Binary Decision Diagrams* (BDDs) [12] to represent feature expressions.

Consider the execution trace s_0,T?,s_1,N!,s_2,F!,s_0 for a system whose current configuration contains only feature GPS.

Our algorithm first processes environment action $T?$, which triggers transition t1 that has destination state s_1 and a weight of 0. Thus, the algorithm updates data structure L to comprise a single tuple that has feature expression $\top$, representing all configurations, system state s_1, and an accumulated reward of 0: $< \top, s_1, 0 >$. The next transition, s_1,N!,s_2, contains a system action, so the algorithm identifies all the transitions that originate from s_1, and are compatible with feature expression $\top$ – that is transitions t2 and t3. Our algorithm determines compatibility by performing a call to satisfiability (SAT) solver *MiniSat* [37] to check whether the corresponding boolean formula ($\gamma \wedge \gamma(t)$) is satisfiable or not. For each one of those two transitions, the algorithm computes a tuple of feature expression, resulting system state, and accumulated reward: $< GPS, s_2, 5 >$ for transition t2 and $< CV \wedge \neg GPS, s_3, 2 >$ for transition t5. The algorithm attempts to merge these two tuples, but is unable to do so as they have different system states and accumulated rewards. To process the next action F!, which is a system action, the algorithm identifies all system transitions that either originate from s_2 and are compatible with feature expression GPS, or that originate from s_3 and are compatible with feature expression $CV \wedge \neg GPS$. These are transitions t5 and t4. After the algorithm computes the resulting tuples for those transitions, the custom data structure L comprises triplets $< GPS, 1, 6 >$ and $< CV \wedge \neg GPS, 1, 3 >$. These results suggest that a configuration that satisfies the feature expression $CV \wedge \neg GPS$ will consume 50% less energy than a configuration that consists of only feature GPS. We hypothesize that this information could be useful to a DSPL to decide whether it would be advantageous to reconfigure.

We implemented our family-based algorithm by extending ProVeLines [30], which is a family-based model checker for software product line models. We give a brief overview

of ProVeLines in subsection 2.4.3. We also extended ProVeLines with a *product-based analysis* that performs trace checking on the execution of each configuration of an SPL individually. The product-based approach generates each product and then performs a separate complete simulation of each product over the observed trace.

As mentioned earlier, our algorithm uses algorithm uses *Ordered Binary Decision Diagrams* (BDDs) [12] to encode the boolean formulas that represent feature expressions. Specifically, our implementation of the algorithm uses the *Colorado University Decision Diagram* (CUDD) [100] to manipulate and implement BDDs. Additionally, to determine whether a boolean formula is satisfiable or not, our algorithm performs calls to satisfiability solver *MiniSat* [37] .

The family-based algorithm takes time $O(|Trace| * |T| * 4^{|N|})$, where $|Trace|$ is the size of the trace, $|T|$ is the number of transitions in the FTS, and $|N|$ is the number of features in the DSPL; This includes the complexity of the SAT calls ($O(2^{|N|})$ per call). The product-based trace-checking approach takes time $O(|Trace| * |T| * 2^{|N|})$. As we mentioned earlier, the potential reduction in execution time are not based on an improvement in the worst-case execution time but instead on sharing of partial-analysis results.

3.4 Evaluation

We assess the efficiency of our analysis technique on three subject systems taken from the literature: a tele-assistance system, a small e-commerce system, and an elevator system. We compare the performance of our family-based algorithm against a product-based analysis that analyzes each configuration individually. We implemented the product-based trace checking analysis by extending *ProVeLines*, which is the same tool that we extended to implement the family-based analysis. We discuss hypotheses for why our family-based analyses performs better on some systems than on others.

3.4.1 Subject Systems

The first case study is a tele-assistance system (taken from [106]). The tele-assistance system monitors a patient with a chronic condition to remotely manage his condition. The system collects vital parameters from the patient and sends those parameters to a medical-analysis service. Based on the results of the medical analysis, the system either calls an alarm service to dispatch help to the patient or maintains/modifies the current medication dosage being given to the patient. Additionally, the system allows the patient to press

a button to call an alarm service. The system can call three different medical analysis services and three different alarm services. Each one of those services has a different cost and cost is the quality of interest. We manually translated a model of this system into a version of Promela (Featured Promela – see subsection 2.4.2) for a brief overview of it) that is extended with support for features [26]. This system has six features that can form nine configurations.

A second case study is an e-commerce application taken from [46]. The application allows users to compare prices of products found in a local store with prices of those products on the web and in nearby stores. The application provides alternative implementations for recognizing the barcode of products and for determining the location of the user. It also provides optional features such as sorting results by distance to the user's location. Each implementation of a feature is annotated with the amount of time it would take to execute it. We manually translated a model of this system into a version of Promela (Featured Promela – see subsection 2.4.2) that is extended with support for features [26]. This application has seven features that can form 24 configurations.

A third case study is a model of an elevator system developed by Plath and Ryan [93] and extended by Classen *et al.* [27]. The elevator has features such as *Park*, which sends the elevator to a specific floor when idle, and *Shuttle*, which makes the elevator move to its target floor without stopping. The elevator system exhibits more functional variability than the other two systems. We use a version of the elevator system that has two passengers and four floors. The elevator model has eight features and 256 configurations.

3.4.2 Experimental Setup

We generated 20 random traces for each subject system. We set the length of the traces at 20,000 transitions for the tele-assistance system and at 150,000 transitions for the e-commerce application.

The elevator system has much more nondeterminism than the other two systems, which means that our trace checking has to keep track of a much larger number of possible system states and takes longer to analyze the system. Thus in the elevator case study, we used a much smaller trace that consists of a sequence of 10 target floor choices (for each of two passengers) — that is, our trace records the floors that each of the two passengers will request when they enter the elevator up to ten times per passenger.

For each subject system, we executed our family-based algorithm and the product-based approach 20 times on each trace and recorded the time taken by each execution.

Table 3.1: Average time taken by product-based, family-based, and abstract family-based trace checking.

System	# features	# configurations	# states	Product-based analysis (seconds)[1]	Family-based analysis (seconds)[1]	Family-based analysis with abstraction (seconds)[1]
Tele-Assistance	6	9	51	2.22 ± 0.02	2.56* ± 0.03	1.60* ± 0.02
E-Commerce	7	24	15	4.15 ± 0.36	3.73* ± 0.40	0.54* ± 0.07
Elevator	8	256	2^{17}	69.81 ± 31.03	12.33* ± 3.18	12.33* ± 3.18

[1]mean ± standard deviation
* Denotes a statistical significant difference compared to the product-based analysis (based on a t-test with p=0.05).

We executed both analyses on a Macbook Pro with 8 GB of RAM, and with an 2.7 GHz Intel Core i5 processor.

3.4.3 Results and Discussion

Table 3.1 shows the average and standard deviation for the time taken by our family-based analysis and by the product-based analysis for each one of the three subject systems.

Our family-based trace-checking analysis is 5.7 times faster than the product-based analysis on the elevator system, and is 1.1 times faster on the e-commerce system, but it is 1.2 times slower for the tele-assistance system. For both the tele-assistance system and the e-commerce system, we observe little to no reduction in execution time for family-based analysis — which is somewhat surprising given the savings observed in other family-based analyses (e.g., [27, 26]). The elevator's analysis time has a large overall standard deviation, but the standard deviation of the time to analyze each individual trace (20 times) is very low (not shown). This suggests that each trace causes the trace checking algorithm to explore a significantly different number of configurations and states.

In the e-commerce and the tele-assistance systems, every feature directly impacts the transitions' weights, and most transitions exhibit different weights for many configurations. This could explain why, for those two systems, most configurations would generate different accumulated rewards in the analyzed traces and the family-based would perform poorly on them. In contrast, in the elevator system only transitions that cause the elevator to move between floors have a non-zero weight. We observed that for the elevator system many configurations would share the same accumulated reward in the analyzed traces. Thus, we

hypothesize that the sharing of accumulated reward values across different configurations is the key aspect that permits a family-based trace-checking algorithm to obtain a large speed-up when analyzing a DSPL. From analyzing these results, we hypothesize that our analysis will perform well on other systems that have only a few different values for their transitions' weights.

3.5 Data Abstraction

Family-based analyses typically outperform product-based analyses, because they can reuse partial-analysis results that apply across multiple products. When analyzing quality-of-service values, different products might result in quality-of-service values (accumulated rewards) that differ by small amounts, while being identical in every other aspect (e.g., same resulting state), which prevents a family-based algorithm from reusing partial-analysis results across those products. However, for some systems, it may be acceptable to disregard small differences in quality-of-service values when making reconfiguration decisions. Thus, we use data abstraction on the value of transition weights to reduce the number of weights and test our hypothesis that few distinct weights in the model leads to an improved performance of the family-based analysis.

Specifically, we categorize transitions into those with a high weight (e.g., high-energy consumption in the example DSPL) and those with a low weight (e.g., low-energy consumption in the example DSPL). Our algorithm then calculates the number of high-weight and low-weight transitions that each set of configurations would have executed (i.e., lines 8 and 12 of our algorithm tracks the number of high and low cost transitions executed instead of tracking the accumulated weight of executed transitions). Our algorithm is then able to group two configurations together if they would have executed the same number of high-weight and low-weight transitions (lines 18-20), even if they would have resulted in slightly different concrete accumulated rewards.

We applied data abstraction to the three case studies in the following manner. For the tele-assistance system we manually partitioned transitions into those with low-cost, high-cost, or zero-cost. For the e-commerce application, we manually classified features (represented as transitions) into those that were fast and those that were slow. For the elevator system, we set the cost of moving the elevator one floor to one unit, so we do not use any data abstraction in that study.

The abstraction of the transitions' weights into either Zero, Low, or High gives rise to an abstract product. An abstract product, denoted p^A, consists of the weighted transition

system associated with product p, but with each one of the transition's weights replaced by either Zero, Low, or High. The accumulated cost of executing an abstract product P^A over a specific trace is then given by the number of high-cost transitions executed and the number of low-cost transitions executed.

3.5.1 Soundness of Data Abstraction

The set of possible concrete costs of executing a transition $t \in T$ comprises all natural numbers used as weights in the FTS; we call this set C_T. The set of possible concrete values for the accumulated cost of executing a trace π comprises all natural numbers; we call this set C_{Acum}. The usual operations of $=, <, \leq$ and $+$ are available between members of these concrete sets.

The set of possible abstract costs of executing a transition $t \in T$ comprises the values Zero, Low, or High, which we denote as the set A_T. The accumulated abstract cost of executing a trace π is represented by a tuple, $a =< High,\ Low >$, that consists of a pair of natural numbers representing the number of high-cost and low-cost transitions executed in a trace. We call the set of all possible tuples A_{Acum}. Given a tuple $a =< High,\ Low >$, we use the notation $a.High$ to reference the first element of tuple a and $a.Low$ to reference the second element of a.

An abstraction function, denoted α, maps concrete values (elements of C_t) to one of three abstract values: $\emptyset$, Low, or High. The minimum concrete value that is mapped to Low is denoted LLB and the maximum is denoted LUB. The minimum concrete value that is mapped to High is denoted HLB, and the maximum is denoted HUB. We note that $\emptyset \leq LLB \leq LUB \leq HLB \leq HUB$.

We define an abstract addition operator, $+_A$, that adds an element of A_{Acum} and an element of A_t, to produce an element of A_{Acum}:

$$\begin{array}{lcl} a_1 +_A\ 0 & = & a_1 \\ a_1 +_A\ Low & = & < a_1.High,\ a_1.Low + 1 > \\ a_1 +_A\ High & = & < a_1.High + 1,\ a_1.Low > \end{array}$$

We define an abstract comparison operator, denoted $<_A$, that compares two abstract accumulated costs, a_1 and a_2, to determine whether a_1 is less than a_2, a_2 is less than a_1, or that a_1 and a_2 are incomparable:

$a_1 <_A a_2$ If $(a_1.High * HUB + a_1.Low * LUB) < (a_2.High * HLB + a_2.Low * LLB)$
$a_1 >_A a_2$ If $(a_1.High * HLB + a_1.Low * LLB) > (a_2.High * HUB + a_2.Low * LUB)$
$a_1 \equiv_A a_2$ Otherwise

Additionally, we define a concretization function, denoted δ, that maps abstract accumulated costs to the set of possible corresponding concrete accumulated costs:

$$\delta(a_1) = \{x \in N \,|(a_1.High * HUB + a_1.Low * LUB) \geq\ x \geq (a_1.High * HLB + a_1.Low * LLB)\}$$

We should note that we have simplified the concretization function by making it output a contiguous set, which is an over-approximation of the set of possible concrete values that can correspond to an accumulated cost.

We can now proceed to prove the soundness of the data abstraction that we have used in this Chapter.

Theorem 1 *Let a_1 be the accumulated abstract cost incurred by abstract product p_1^A when executed on an input trace π, and let a_2 be the accumulated abstract cost incurred by another abstract product p_2^A when executed on the same trace π. Let c_1 be the accumulated concrete cost incurred by the original product p_1 when executed on the same trace π, and let c_2 be the concrete cost incurred by the original product p_2 when executed on the same trace π. If $a_1 <_A a_2$, then $c_1 < c_2$.*

Proof:

Assume that $a_1 <_A a_2$ is true.

Since $c_1 \leq \max \delta(a_1)$, then $c_1 \leq (a_1.High * HUB + a_1.Low * LUB)$.

Since $c_2 \geq \min \delta(a_2)$, then $c_2 \geq (a_2.High * HLB + a_2.Low * LLB)$

Since $a_1 <_A a_2$, then $(a_1.High * HUB + a_1.Low * LUB) < (a_2.High * HLB + a_2.Low * LLB)$.

$\therefore c_1 \leq (a_1.High * HUB + a_1.Low * LUB) < (a_2.High * HLB + a_2.Low * LLB) \leq c_2$

$\therefore c_1 < c_2$

3.5.2 Results and Discussion

Our family-based algorithm, with data abstraction, performs better and is faster than the product-based analysis for all three case studies. Specifically, our algorithm is 1.39 times faster than the product-based approach[2] for the tele-assistance system and 7.7 times faster for the e-commerce system (for the elevator system no data abstraction is possible as weights are already abstracted to two values: zero versus one). Because we are comparing the arithmetic means of two normal distributions, we use the t-test to determine whether their differences are statistically significant. Based on a t-test (p=0.05) these differences are statistically significant for all three systems. These results suggest that data abstraction can speed-up family-based analysis of DSPLs.

In the above description, we have only considered analysis of a single quality attribute. Our approach can be extended to multiple quality attributes by introducing a vector of weights for each transition. However, when analyzing multiple quality attributes simultaneously aggressive abstraction would be necessary as otherwise every configuration could provide a slightly different vector of accumulated rewards, and family-based analysis is less likely to outperform product-based analysis.

3.6 Related Work

A class of approaches [21, 90, 91] associate with each feature or adaptation a static value for quality of service or a static value for cost or resource usage. These values represent a feature's entire contribution to the system's quality of service in these analyses. Whether These values apply (or not) depends on whether (or not) the feature is active in a configuration. Approach [90, 91] uses a genetic algorithm to search for an optimal configuration that optimizes for quality of service within current resource constraints. At runtime, the availability of resources is monitored, and a change in resource constraints may trigger a new runtime search for a more optimal configuration, in terms of the sum of the feature's contribution to the quality of service (i.e., using a coarse-grained model of a configuration's quality of service). In contrast, our models of features record how a feature's contributions to quality of service can vary with the features' behaviour and actions (i.e., depending on which transitions are executed) during a feature's execution. This allows for a more accurate assessment of a system's quality that takes into account not only changes of

[2]We compare against the concrete product-based trace checking algorithm because the abstraction on weight values has no effect on the execution time of the product-based trace checking algorithm.

the operating environment but also changes in the executing system, and how a feature's contribution to a system's quality-of-services changes as the environment changes.

In approaches that use Discrete-Time Markov Chain (DTMC) models with rewards [13, 41, 47], some transition probabilities and rewards are represented as uncertain variables, which are updated at runtime in response to changes to the system's configuration or environment. Calinescu et al. [13] present a self-adaptive system that periodically updates such a model at runtime and verifies, using computationally-intensive probabilistic model checking, whether its quality-of-service requirements will continue to be met. If not, it exhaustively analyzes each one of its configurations to reconfigure to the optimal one. The work of Gerasimou et al. [44] improves such analysis performance by caching previously seen configurations and preemptively analyzing configurations that will likely be encountered. To reduce runtime analysis time, Filieri et al. [41, 42] compute, at design-time, a single polynomial expression (consisting of constants and variables) that can represent the quality of service (e.g., reliability, energy consumption) of any system configuration and environment; this expression can be efficiently re-evaluated at runtime when estimates for the variables's values are updated. Similarly, Ghezzi *et al.* [47] model a DSPL as a sequence diagram (which is transformed to an equivalent DTMC) where features can introduce new transitions and are assumed to be independent. They leverage the analysis of Filieri et al. [42] to compute a polynomial for the base DSPL and for each feature, which allows them to efficiently evaluate the quality of service of any configuration at runtime. Their approach still uses a static-cost model of the quality-of-service of a DSPL by using a static value of quality of service per feature, representing the feature's entire contribution to the system's quality of service. In contrast, our work does not assume features to be independent, permits transitions to depend on multiple features, and permits the execution of transitions to impact the estimated quality-of-service values.

Proactive adaptation [80] uses nondeterminism to model the latency of each adaptation, the uncertainty about the future states of the environment, and the choices among the different adaptations. At runtime, the probabilistic model checker PRISM [72] is used to decide which adaptation maximizes the overall utility over a small look-ahead period. In contrast, instead of attempting to predict the future, our approach uses the recent past as a proxy to the environment's near-future behaviour and selects the configuration that would have performed the best in the recent past. For scalability, Moreno et al. [79] adapt an approximate optimization algorithm to find near-optimal configurations much faster. There exist different schools of thought about how to best predict the future: probabilistic models versus the recent past. The usefulness of the former approach depends on the ability to create probabilistic models. At the least, the recent past can be used while the probabilistic model is being learned.

To decide whether to reconfigure a system, existing approaches maintain and update a model of the system's environment at runtime, and then periodically analyze — using computationally intensive techniques such as probabilistic model checking — the expected quality of service that each configuration would provide [13, 80]. The efficiency of these analyses is a major concern as they are performed at runtime and they analyze a large number of configurations. Researchers have improved the scalability of these analyses by performing part of the computation in advance and sharing partial-analysis results across multiple configurations [42, 81], by performing an approximate analysis that returns the best configuration found (so far) after a fixed amount of time has elapsed [79], and by identifying near-optimal configurations [44]. Despite these optimizations, quickly analyzing the quality of service that every configuration would provide remains challenging. Alternatively, other approaches use static-cost models that estimate the impact that each adaptation [14] or each activation/deactivation of a feature [91] would have on the quality of service. These simpler models can be analyzed much faster, however they do not capture how the environment affects the performance of the system nor the interactions between multiple features.

3.7 Conclusions

We have presented a family-based trace-checking analysis that estimates the quality of service that each configuration of a DSPL would provide. We have assessed the efficiency of such a family-based analysis on three case studies from the literature. Without data abstraction, our family-based analysis faster than analyzing each configuration individually in only one system because of the lack of sharing of partial-analysis results across configurations due to slight differences in quality-of-service performance. With a simple data abstraction, the efficiency of our family-based analysis improves substantially and our analysis is between 1.4 and 7.7 times faster than analyzing each configuration individually, an improvement which could be significant given that the analysis is to be performed at runtime.

Chapter 4

Long-term Average Cost in Featured Transition Systems

As mentioned in Section 2.1, the techniques for analyzing software product lines can be categorized into family-based or product-based A family-based analysis is performed on a single model that represents all the possible products in an SPL. Thus, family-based approaches avoid some of the redundant computations that are inherent in product-based analyses; but they require that standard analysis algorithms be adapted to accommodate variability in the SPL model.

In this chapter, we focus on an analysis of quality attributes, called *limit average*, which computes a long-term average of a quality attribute for a product [68, 107]. We adapt the limit-average algorithm in order to perform a family-based analysis that computes the limit average of a quality attribute for every product in a software product line that is modelled as a weighted featured transition system.

Our contributions in this chapter include:

- A family-based algorithm that analyzes a model of an SPL that models every product and simultaneously computes the limit average for a quality attribute for every product.
- An implementation of the family-based algorithm.
- An evaluation of the speed-up of our family-based approach versus the product-based approach.

4.1 Background

4.1.1 Limit-Average Cost

The *limit-average cost* [107] expresses the average cost of a quality attribute over the steps in a single infinite execution of a weighted transition system. Thus, if the weight of an execution step represents the consumption of a resource during that execution step, then the limit average represents the long-term average rate of resource consumption per step along a single (infinite) execution trace.

Definition 8 *Given an infinite execution trace* $\pi = s_0\alpha_1 s_1\alpha_2 \ldots$ *of a weighted transition system, consider 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 cost *for trace* π *is then defined to be*[1]

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

The *maximum limit average* [107] cost of a weighted transition system is the maximum cost among the limit average costs for all of its execution traces. By computing the minimum and maximum limit-average cost of a weighted transition system whose weights represent, for example, energy consumption, we obtain the best-case and worst-case long-term rates of energy consumption for the system. In contrast, the worst-case execution cost of a weighted transition system comprises the sum of the maximal costs (weights) along an infinite execution trace of the system, which could be infinite or could be impacted by one-time costs that are irrelevant in the long term. A maximum limit-average cost over an infinite execution trace is impacted only by those costs that are incurred an infinite number of times within the trace. Thus, the maximum limit-average cost is better suited to represent the long-term cost of a continually executing system, where one-time costs have minimal impact over time and can be ignored.

The **maximum limit-average cost** can be computed by a two-phase algorithm [107]: the first phase computes the set of strongly connected components; then the second phase identifies for each strongly connected component the cycle with the highest average-weight. Finally, **the average weight of the highest-average-weight cycle that is reachable from the initial state** is the **maximum limit-average cost** for the weighted transition

[1] We note that we use the limit of the infimnum of the formula instead of the limit of the formula to ensure that such limit always exists.

system. In the rest of this chapter, we will interchangeably use the terms "average weight of the highest-average-weight cycle that is reachable from the initial state" and "maximum limit-average cost".

4.1.2 Strongly Connected Components (SCCs)

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 [33]. Any cycle in a graph will be contained within an SCC. Hence by searching for the maximum average-weight cycle within each SCC of a graph, we obtain the maximum average-weight cycle of the full graph [107].

The standard algorithm [33, 96] for computing the SCCs of a graph $G = (V, E)$ proceeds as follows:

First, the algorithm performs a depth-first search of the graph and computes for each node its *finishing time* in the depth-first search. We state the definition of the finishing time of a node in a depth-firsth search below.

Definition 9 *Let $G = (V, E)$ be a directed graph where V is a set of nodes and E is a set of directed edges between nodes. In a depth-first search exploration of G the* ***finishing time****, $FinishingTime(v)$, of a node v is the temporal order in which the node v-plus all of the nodes that a depth-first-search from v can reach-has been fully explored.*

Thus, the first node that is finished being explored (a leaf node) has a finishing time of 1, and the last node that is finished being explored has a finishing time of $|V|$. The finishing time of a node, $FinishingTime(v)$, ranges from 1 to $|V|$.

Second, the algorithm then processes the nodes in decreasing finishing times. It starts with the node v whose $FinishingTime(v) = |V|$ 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). This set of nodes corresponds to an SCC [33].

Finally, 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. Different depth-first searches may result in different finishing-times, but this does not affect the final set of SCCs. The algorithm takes time $O(V + E)$.

In order to compute the maximum limit-average cost of a weighted graph, one needs to identify the highest average-weight cycle within each SCC of the graph. This is usually

done using Karp's algorithm [68]. For each SCC, Karp's algorithm chooses an arbitrary "source state" s_{start} and then iteratively computes a function F that associates with each state v in the SCC and each path length $k \in \{0, \ldots, (n-1)\}$ (where n is the size of the SCC) the maximal weight of a path of length k from s_{start} to v. By Karp's theorem [68], the weight of the maximal average-weight cycle is given as:

$$\max_{v} \min_{k<n} \frac{F\Big(n, v\Big) - F\Big(k, v\Big)}{n-k} \tag{4.1}$$

In this formula, the expression $F\Big(n, v\Big) - F\Big(k, v\Big)$, when evaluated on the maximizing state v and the minimizing path-length k, represents the total cost of the maximal average-weight cycle in the SCC from state v to itself; and the denominator $n-k$ represents the length (number of edges) of the maximal average-weight cycle (also evaluated on the maximizing v and minimizing k). Karp's algorithm to compute the maximal average-weight cycle takes time $O(V \times E)$.

The above algorithm applies to a simple weighted graph. In the rest of this Chapter, we extend the algorithm to apply to a weighted featured transition system.

4.2 Motivating Example

Figure 4.1 shows a small weighted featured transition system for a combined taxi and shuttle service. The shuttle service (feature S) picks up passengers from several pickup locations and delivers them to the airport, or picks up passengers at the airport and drops them off at several city locations. The taxi service (feature T) can transport passengers between city locations, and not just to and from the airport. There are three pickup and three release locations in the city, one of which is accessible only to vehicles that have an extra license (feature L). The weights on the transitions show their cost: positive numbers are income and negative numbers are expenses.

This example SPL has three features, S, T and L, giving rise to eight products (as the feature model allows all possible products in this example): $\emptyset$, $\{L\}$, $\{S\}$, $\{T\}$, $\{L, S\}$, $\{L, T\}$, $\{S, T\}$, and $\{L, S, T\}$. An interesting problem is to compute the maximal income for each product; the maximum limit-average cost is an approximation of this maximal income.

A product-based analysis reveals that each product has exactly one SCC that can be reached from the initial state (where the initial state is Airport-P: the vehicle is at the airport pickup).

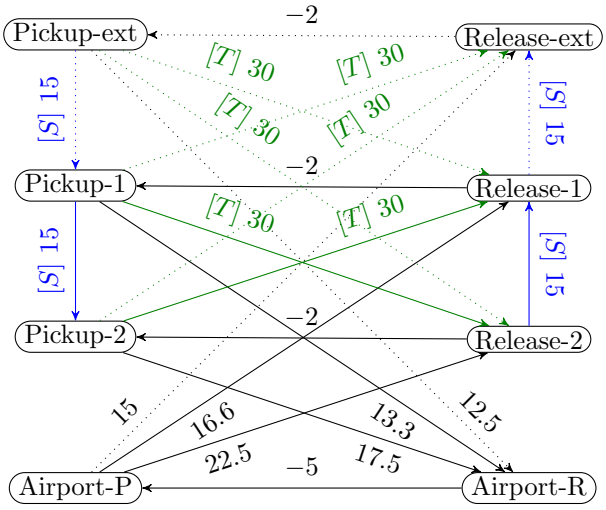


Figure 4.1: Taxi-shuttle example. In addition to the feature guards shown, all *dotted* transitions are guarded by the feature L. States refer to the current location of the vehicle.

In product $p = \emptyset$ there are two cycles:

$$\text{Airport-P} \to \text{Release-1} \to \text{Pickup-1} \to \text{Airport-R} \to \text{Airport-P} \tag{4.2}$$

$$\text{Airport-P} \to \text{Release-2} \to \text{Pickup-2} \to \text{Airport-R} \to \text{Airport-P} \tag{4.3}$$

Their average weights are 10.38 and 12.17, respectively (rounded to two decimal places), hence cycle (4.3) between the airport and city location 2 provides the maximal income.

In $p = \{L\}$ there are three cycles: the cycles listed above plus a third:

$$\text{Airport-P} \to \text{Release-ext} \to \text{Pickup-ext} \to \text{Airport-R} \to \text{Airport-P} \tag{4.4}$$

But the average weight of the steps in the third cycle is only 10.30, so cycle (4.3) is still the most profitable.

In $p = \{S\}$, there are five cycles: in addition to cycles (4.2) and (4.3) above, there are three other cycles:

$$\text{Airport-P} \to \text{Release-2} \to \text{Release-1} \to \text{Pickup-1} \to \text{Airport-R} \to \text{Airport-P} \tag{4.5}$$

$$\text{Airport-P} \to \text{Release-1} \to \text{Pickup-1} \to \text{Pickup-2} \to \text{Airport-R} \to \text{Airport-P} \tag{4.6}$$

$$\text{Airport-P} \to \text{Release-2} \to \text{Release-1} \to \text{Pickup-1} \to \text{Pickup-2} \to \text{Airport-R} \to \text{Airport-P} \tag{4.7}$$

Their average weights are 11.63, 11.63, and 12.88, respectively; hence for a purely shuttle product, cycle (4.7), which picks up and releases passengers at both city locations, is most profitable.

Similar analyses can be done for the remaining five products. However, a family-based analysis that computes SCCs and maximum average-weight cycles for all products at once would be preferable. We will come back to this example in Section 4.4.

4.3 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 computing SCCs for all products (subsections 4.3.1 and 4.3.2) and then for each SCC in each product it computes the maximum average-cost cycle (subsection 4.3.3).

The computation of SCCs requires exploring the transpose of the WFTS graph in reverse order of the finishing times of states in a depth-first search. In the rest of this chapter, we will use the term finishing time of a state to refer to the temporal order in which the state and all of its descendants are finished being explored in a depth-first search (starting from an initial node s_0). As a first step in subsection 4.3.1, we compute and store a mapping from sets of products to finishing times of states in a depth-first search. An example of such a mapping is shown in Figure 4.3, where node s_0 has the highest finishing time if feature G or A is present and node s_2 has the highest finishing time otherwise. In subsection 4.3.2, we use such a mapping to compute the SCCs in an adapted version of the standard algorithm to compute SCCs. This algorithm explores a tree like the one in Figure 4.3 in a breadth-first manner and computes a featured set of SCCs for each set of finishing times of nodes, producing a tree like the one shown in Figure 4.4; the latter represents the set of SCCs that different feature combinations induce (we will explain such a figure in subsection 4.3.2).

In order to illustrate the family-based SCC computation, we introduce a tiny SPL, modelled as a WFTS in Fig. 4.2, of an arbiter granting access to a shared resource. In one product, representing the basic system without features, access is granted only after a request has been received. A second product always grants access, whether or not a request exists (represented by transitions labelled with feature expression G). A third product alternates between granting access and not granting access (represented by transitions labelled with feature expression A).

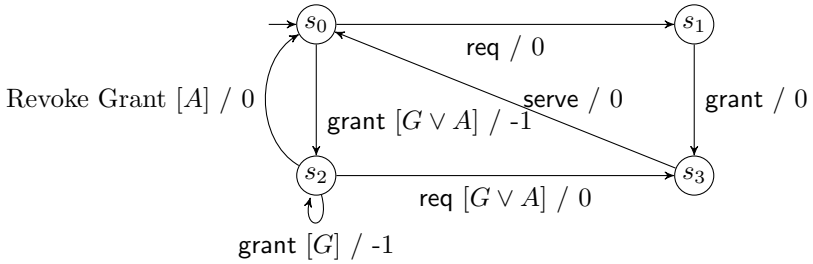


Figure 4.2: WFTS which implements several grant/ request scenarios.

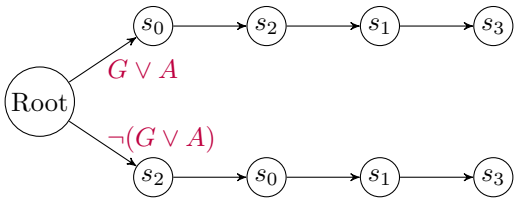


Figure 4.3: Featured finishing-times tree for the FTS from Fig. 4.2. It maps feature expressions to the finishing times of nodes. In the lower path state s_2 has a higher finishing time than initial state s_0, so s_2 is unreachable in the products associated with such path.

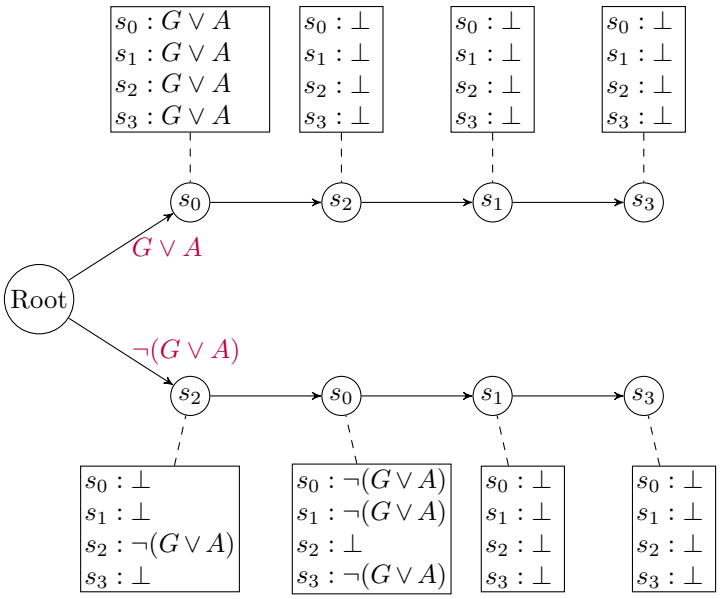


Figure 4.4: Featured SCC tree for the FTS of Fig. 4.2. As in Fig. 4.2, state s_2 and its associated SCC are unreachable in all the products associated with the feature expression $\neg(G \vee A)$.

Each of these products satisfies the functional requirements of the system, namely that a request always leads to granted access. However the user may prefer one solution over another because of the solution's qualitative properties; for example she might want to minimize the number of unnecessary grants. This preference is encoded as weights on the transitions, such that a penalty of -1 is given every time a grant is given without having been requested.

4.3.1 Featured Finishing Times

The second phase of the standard algorithm for computing the SCCs of a graph expects as input the finishing times of the graph's nodes in a depth-first search starting from an initial node s_0. However, a featured transition system represents a *set* of transition systems, each with a different set of transitions, which can give rise to different finishing times for its states. For example, the basic product in Fig. 4.2 (without feature A nor G) with initial state s_0 would have the following finishing times for each state[2]:

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

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

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

As a first step, given a weighted featured transition system, we construct a *tree* that represents the finishing times of states for every product.

In such a *featured finishing-times tree*, each path 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 the 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 conjunction of the feature expressions along the path from the root to the node. If the path goes all the way from root to a leaf node, then the set of products satisfying the path's feature expression (i.e., the conjunction of the feature expressions along the path from the root to the node) share an identical set of finishing times in a depth-first exploration starting from the initial state.

[2]In general, the "finishing times" of the nodes in a depth-first exploration of a graph are not necessarily unique – they will depend on the order in which transitions and states are explored when multiple transitions/states can be chosen for exploration. However, our algorithm always chooses the transition/state to explore next in the same fixed order when multiple states/transitions are available to explore – so this yields unique "finishing times'.

For example, the WFTS from Fig. 4.2 gives rise to the featured finishing-times tree shown in Fig. 4.3. This tree assigns one set of "finishing times" to all products that contain either feature G or A, and another set of "finishing times" to products that contain neither feature. In products that contain either feature G or A, state s_0 has a finishing time of $|S| - d + 1 = 4 - 1 + 1 = 4$, state s_2 has a finishing time of $4 - 2 + 1 = 3$, state s_1 has a finishing-time of $4 - 3 + 1 = 2$, and state s_3 has a finishing time of $4 - 4 + 1 = 1$.

Definition 10 *Let fts =(S, I, A, T, d, γ) be a featured transition system. A* ***featured finishing-times tree*** *for fts consists of a tree $TF = (V, E)$ of height $n = |S|$ where V is a set of vertices and E is a set of edges, 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. Let squared brackets around a feature expression (e.g., $[\![G \vee A]\!]$) denote the set of software configurations that satisfy the feature expression. We recall from Definition 1 that px is the set of all valid software configurations. The tree TF 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 configurations associated with the outgoing edges labels 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 the path's node at level i such that the product p is contained within the set of configurations represented by the conjunction of the feature expressions along the edges of the path: $\forall p \in px, i \in \{1, \ldots, n\} : \exists!$ path $v_0, \ldots, v_i : p \in \bigcap_{j=0}^{i-1} [\![\ell_e((v_j, v_{j+1}))]\!]$.*
- *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 featured finishing-times tree is built in two phases. In the first phase (performed by Algorithm 2), a featured depth-first search explores all states of an FTS and computes a temporal ordering for when a state and all its DFS descendants are explored, depending on feature expressions. The second phase (shown in Algorithm 3) uses this information to construct a featured finishing-times tree in a breadth-first manner.

In Algorithm 2, unlike in a standard depth-first search algorithm, states are not marked as visited with a boolean flag but instead with a feature expression representing *under which set of products* have they been visited. Hence Algorithm 2 stores and updates an array

Alg. 2: Depth-first search for a Featured transition system

```
1   Global: FinishingTimes: Array[state] of a set of < feature expression,
finishing time>
2   Procedure DFS-FTS (fts)
3   Input: fts = ( S, I, A, T, d, γ) : A featured transition system
4   Local: Unexplored: Array[state] of feature expression (represents the products
           for which a state not yet been explored)
5   begin
6       for each u ∈ S
7           Unexplored[u] ← ⊤
8           FinishingTimes[u] ← ∅
9       time ← 0
10      for each u ∈ S
11          if Unexplored[u] is satisfiable then
12              DFS-FTS-Visit(fts, u, Unexplored[u], Unexplored, time)
13          end-if
14  end
15
16  Procedure DFS-FTS-Visit(fts, u, λ_u, Unexplored, time)
17  Input: fts: A featured transition system
18  Input: u: A state to explore
19  Input: λ_u: A feature expression indicating for which products to explore state u.
20  Input/Output: Unexplored: Array[state] of feature expression (represents
                  the products for which a state not yet been explored)
21  Input/Output: time: A counter that is incremented as each vertex is explored
22  begin
23      Exploring ← Unexplored[u] ∧ λ_u
24      Unexplored[u]← Unexplored[u] ∧¬λ_u
25      for each t=(u, v, λ_t) ∈ T
26          NextFExp ← λ_t ∧ λ_u
27          if (Unexplored[v] ∧ NextFexp) is satisfiable then
28              DFS-FTS-Visit(fts, v, NextFexp, Unexplored, time)
29          end-if
30      time ← time + 1
31      FinishingTimes[u].insert(<Exploring, time>)
32  end
```

Unexplored of boolean formulas: Each array index of *Unexplored* represents a unique state in the FTS, and the state's corresponding array element is a feature expression representing the products for which the state has not been explored.

Algorithm 2 starts by initializing the array *Unexplored* to true (all products) for each state (lines 6-8). It also initializes the global array *FinishingTimes* to the empty set for each state (lines 6-8). This array will later be used to store a set of pairs of feature expressions and finishing times for each state. The algorithm then initializes a counter, denoted *time*, that will be incremented each time a state is finished being explored (line 9). The algorithm then iterates over all states, and for each state that has not been fully explored it calls the subroutine DFS-FTS-Visit, passing as parameters that state and the feature expression representing that state's set of unexplored products, a reference to array *Unexplored*, and a reference to variable *time* (lines 10-12).

The subroutine DFS-FTS-Visit starts by updating (reducing) the set of unexplored products for its given state u (lines 23-24) because it will start exploring (i.e., perform a new depth-first search) from the input state u, with products that satisfy the input feature expression. This way the algorithm ensures termination by preventing the state u from being explored again with those products. Then it iterates over each outgoing edge from the state u and checks if there are products

1. that satisfy the input feature expression,
2. that satisfy the feature expression associated with the outgoing transition, and
3. in which the destination state has not been explored (i.e., if $\lambda_u \wedge \lambda_t \wedge$ Unexplored$[v]$ is satisfiable; We note that our algorithm checks if such a formula is satisfiable by performing a call to satisfiability (SAT) solver *MiniSat* [37].) (lines 26-27).

If so, then it recursively calls itself to explore the destination state. Finally, once all outgoing edges from the state u have been explored, it increments the *time* counter and sets the finishing time for the input state u and feature expression $Exploring$ to the current *time* counter (lines 30-31) – this is achieved by inserting the tuple $< Exploring, time >$ to the set associated with state u in array $FinisingTimes$. This feature expression represents the products for which state u and all its descendants have been fully explored at timestep *time*. The array $FinishingTimes$ contains the featured finishing-times for each state in the FTS and is the output of Algorithm 2.

Once the feature-based depth-first ordering of states has been computed, this data can be used to construct the featured finishing-times tree for the FTS. This is done by iterating

over the states in reverse order of their finishing-times, recursively adding a new child to a tree node whenever a new pair (s, λ) is found for which the conjunction of the following formulae is satisfiable (lines 18-21 in Algorithm 3):

1. λ,
2. the negation of the disjunction of the feature expressions on the edges from the tree node to its other children, and
3. the feature expression associated with the path from the root to the current tree node

The check is performed to ensure that each path in the featured finishing-times tree from the root to a tree node's child nodes represents a disjoint nonempty set of software products. The procedure is shown in Algorithm 3 and two auxiliary procedures are shown in Algorithm 4.

The algorithm starts by initializing a local variable, denoted *Tree*, with an empty root node and adding the root node to a queue, denoted Q, of tree nodes to explore (lines 6-9). The algorithm then enters a loop where it processes tree nodes from the queue and computes all their children (lines 11-24). Finally, the algorithm returns a fully populated featured finishing-times tree.

For a given tree node, denoted *TreeNode*, the algorithm identifies all the children of *TreeNode* by iterating over the finishing-times that are lower than *TreeNode*'s maximum finishing-time number in decreasing order (lines 16-24). The goal of lines 16-24 is to identify which states will exhibit a finishing time that is one unit less than *TreeNode*'s finishing time and for which products they will exhibit such value for its finishing time. The algorithm searches for pairs of states and feature expressions $(u, \lambda) = \text{FinishingTimesInverse}(i)$ such that the feature expression (λ) combined with the negation of all other edges leaving the tree node is satisfiable (line 16-24). We note that our algorithms determine if such formula is satisfiable by performing a call to satisfiability solver *MiniSat* [37]. By incorporating the negation of the feature expressions of all other edges leaving the node into the feature expression (which is temporally stored in variable *NotChildren*), this algorithm ensures that the sets of products associated with each of *TreeNode*'s outgoing edges are disjoint. If the feature expression is satisfiable, then it adds a new child node to the tree (line 19) and updates the expression representing the negation of all edges leaving the tree node (line 22) and records the new node's *maxFTime* to be j (which represents the finishing-time of state u in all products that satisfy $\lambda \wedge \lambda_1$), so that later when processing this new node only states with a lower finishing-time than it are searched. The algorithm then adds the new tree node to the queue (line 21). After all children for a tree node have been identified

Alg. 3: Building a featured finishing-times tree for an FTS

```
1   Procedure ComputeFeaturedTree(FinishingTimesInverse, MaximumFTime)
2   Input: FinishingTimesInverse: A function that given a number returns the
           state and feature expression associated with it in FinishingTimes.
3   Input: MaximumFTime: The maximum finishing time in FinishingTimes.
4   Output: Tree: A featured finishing-times tree
5   begin
6       Q ← Empty Queue
7       Tree ← New Tree()
8       Tree.root.maxFTime ← MaximumFTime
9       Q.add(Tree.root)
10      while (¬Q.isEmpty())
11          TreeNode ← Q.pop()
12          λ1 ← FeatureExpressionFromRoot(Tree, TreeNode)
13          max ← TreeNode.maxFTime
14          notChildren ← ⊤
15          j ← max -1
16          while (j > 0)
17              u, λ ← FinishingTimesInverse(j)
18              if (λ ∧ notChildren ∧ λ1 is satisfiable) then
19                  NewTreeNode ← CreateNode(Tree, TreeNode, u,
                                             λ ∧ notChildren)
20                  NewTreeNode.maxFTime ← j
21                  Q.add(NewTreeNode)
22                  notChildren ← notChildren ∧ ¬λ
23              end-if
24              j ← j − 1
25      return Tree
26  end
```

Alg. 4: Auxiliary procedures for Algorithm 3

```
1   Procedure CreateNode(Tree, ParentNode, State, λ)
2   Input: Tree: A featured finishing-times tree
3   Input: ParentNode: A tree node
4   Input: State: A state to associate with the new node
5   Input: λ: A feature expression
6   Output: A new tree node
7   begin
8       NewNode ← New Node()
9       ParentNode.addChild(NewNode)
10      Tree.StateLabel(NewNode) ← State
11      Tree.EdgeLabel(ParentNode, NewNode) ← λ
12      return NewNode
13  end
14
15  Procedure FeatureExpressionFromRoot(Tree, Node)
16  Input: Tree: A featured finishing-times tree
17  Input: Node: A tree node
18  Output: A feature expression
19  begin
20      if Node = Tree.root then
21          return ⊤
22      else
23          return Tree.EdgeLabel(Parent(Node), Node) ∧
                FeatureExpressionFromRoot(Tree, Parent(Node))
24      end-if
25  end
```

and added to the queue (i.e., when the expression in line 18 is unsatisfiable), the nodes remaining in the queue are processed (line 11). As we mentioned earlier, our algorithms determine if a formula is satisfiable by performing a call to satisfiability solver *MiniSat* [37].

This algorithm uses two auxiliary procedures: *CreateNode* (called in line 19) and *FeatureExpressionFromRoot* (called in line 12), which are listed in Algorithm 4. The purpose of procedure *CreateNode* is to create and insert a new tree node into a featured finishing-times tree. The purpose of procedure *FeatureExpressionFromRoot* is to compute the feature expression associated with a path in a featured finishing-times tree from the root of the tree to the input node. The procedure outputs the constructed feature expression.

4.3.2 Strongly Connected Components of a Featured Transition System

The goal of the standard algorithm for computing SCCs [33] is to partition the vertices of a graph into a set of strongly connected components (we recall that an SCC is a maximal set of vertices such that each vertex is reachable from any other vertex in that set). In the context of this work, the set of SCCs is different for different software products, which have different sets of states and transitions. In this work, we have adapted the standard algorithm for computing SCCs [33] to compute the SCCs for an FTS. Our adapted algorithm accepts as input a featured finishing-times tree. The output of the algorithm is a set of SCCs for each path in the featured finishing-times tree, which we call a *featured SCC tree*. We use the word *featured* to name this tree because it associates SCCs to software products that satisfy specified feature expressions.

As an example of the algorithm's output, the (very simple) featured SCC tree of the grant/request WFTS is displayed in Fig. 4.4. Its tree structure is the same as the featured finishing-times tree (shown in Figure 4.3), but now the states are labeled with a mapping from $S \rightarrow \mathcal{P}(N)$ that specifies which states are part of the SCC found starting at that state or the empty relationship if no SCC starts at that state. Each rectangle in Fig. 4.4 represents a (featured) SCC (or the empty relationship if no SCC starts at its associated state). For example, the rectangle associated with state s_0 in the upper path lists all states and associates every state with $G \vee A$. This indicates that, for all products that satisfy $G \vee A$, an SCC comprising all states starts at s_0. We note that Figure 4.4 includes an SCC that is not reachable from the initial state: the SCC associated with state s_2 in the bottom path is not reachable from the initial state because state s_2 itself is not reachable from the initial state in the products that satisfy $\neg(G \vee A)$.

Alg. 5: Computing strongly connected components for an FTS given a featured finishing-times tree for the FTS.

```
1   Procedure FeaturedSCCs (fts, Tree)
2   Input: fts = ( S, I, A, T, d, γ): a featured transition system
3   Input: Tree: a featured finishing-times tree
4   Output: RC: A function from tree nodes to featured SCCs
5   begin
6       NodesToExplore ← Empty stack of triplets of <tree node, state,
        feature expression >
7       ReachabilityStack ← Empty stack of mappings of S → 𝔹(N)
8       for each e = (Tree.root, u) ∈ Tree.Edges
9           InPrevSCC ← new array of size |S| of feature
            expressions (initialized to ⊥)
10          λ_start ← Tree.EdgeLabel(e)
11          s_start ← Tree.StateLabel(u)
12          NodesToExplore.push(<u, s_start, λ_start>)
13          ReachabilityStack.push(InPrevSCC)
14          while ¬ (NodesToExplore.isEmpty()) do
15              <y, s, λ >← NodesToExplore.peek()
16              Visited(y)← True
17              if λ ∧ ¬ InPrevSCC(s) is satisfiable then
18                  RC(y) ← VisitDFS-For-SCC(fts, s, λ ∧ ¬ InPrevSCC(s),
                                                 InPrevSCC)
19                  for each s ∈ S
20                      InPrevSCC(s) ← InPrevSCC(s) ∨ RC(y)(s)
21              end-if
22              Choose v in Children(y) with Visited(v)=False
23              if no such v exists then:
24                  NodesToExplore.Pop();
25                  InPrevSCC ← ReachabilityStack.Pop()
26              else
27                  NodesToExplore.push(<v, Tree.StateLabel(v),
                                          λ ∧ Tree.EdgeLabel(y,v)>)
28                  ReachabilityStack.push(CopyOf(InPrevSCC))
29              end-if
30      return RC
31  end
```

The algorithm to compute the featured SCCs is shown as Algorithm 5. In the standard SCC algorithm, a boolean array keeps track of which states (vertices) have already been assigned to an SCC. In the case of computing SCCs for an FTS, which states belong to an SCC depends on which product the SCC is being computed for. Thus, the algorithm uses an array *InPrevSCC*, indexed by states in S, of feature expressions to keep track of the products for which a state has already been assigned to an SCC. Each child node of the root of the featured finishing-times tree is allocated an *InPrevSCC* array, initialized to $\perp$ for every element (line 9) to represent that no state has yet been assigned to an SCC for any software product.

Algorithm 5 starts by successively exploring each outgoing edge from the root of the tree (line 8). Because the set of states already assigned to an SCC change as the algorithm explores different nodes of the featured finishing-times tree, the algorithm uses a stack, denoted *ReachabilityStack*, to push and pop different versions of *InPrevSCC* as the algorithm processes different nodes of the featured finishing-times tree (lines 9, 13, 25, 28). The algorithm uses a second stack, denoted *NodesToExplore*, to keep track of the nodes still left to explore. For each path from the *Tree* root, the algorithm initializes *InPrevSCC* and then adds to *NodesToExplore* a triplet consisting of the path's child node of the *Tree* root, its associated state, and its feature expression label (lines 9-12); the feature expression represents for which products the node's state would exhibit the highest finishing-time in a depth-first exploration.

The algorithm then enters a loop where elements of the *NodesToExplore* stack are processed (lines 14-29), which corresponds to a depth-first exploration of the featured finishing-times tree. At the start of each iteration, a triplet $<y,\ s,\ \lambda>$ of tree node, state, and feature expression respectively, is peeked from the stack (without being popped). The feature expression λ is compared to $InPrevSCC(s)$, which contains the set of products for which the given state has already assigned to an SCC, and if there exist products that satisfy λ and that do not satisfy $InPrevSCC(s)$, then a new featured SCC is computed by calling *VisitDFS-For-SCC* (line 17-18) and storing the result in $RC(y)$, which is a function mapping tree nodes to featured SCCs.

The set of products associated with an SCC for each state, represented by *InPrevSCC*, is then updated with the newly identified SCC (line 19-20). After processing the current tree node, the algorithm looks for a child that has not been explored (line 22). If no such child exists, then the current element is popped from the *NodesToExplore* stack (line 24) and *InPrevSCC* is assigned to an element popped from the *ReachabilityStack*, which would represent the states already assigned to an SCC when the algorithm first encountered tree node u (line 25). Otherwise, a triplet is built for $<$ the child node v, its state label, and the conjunction of the feature expression λ for the path from the *Tree* root to current *Tree*

node y and the feature expression associated with the edge from *Tree* node u to child node $v >$ and is pushed onto the *NodesToExplore* stack (lines 26-27). The algorithm continues processing triplets in the $NodesToExplore$ stack until it is empty and the complete finishing-times tree has been explored.

We recall that in the standard algorithm to compute the SCCs of a graph, the SCCs are identified by successively computing the set of states reachable from a source state u in the transpose of the graph, where the source states are selected by iterating over all states of the graph in decreasing finishing time (considering only the states that have not already been assigned to an SCC). The auxiliary procedure *VisitDFS-For-SCC* computes the set of states that are reachable from a given state s in the transpose of the FTS, parameterized by feature expressions. This is inspired by the "symbolic" reachability algorithm of [26], except that here we exclude from the search the states that have already been assigned to a previous SCCs. The procedure is shown as Algorithm 6.

Algorithm 6 takes as input an initial state, a feature expression, and a featured set of excluded states (i.e., *InPrevSCC*) and computes the featured set of states that are reachable, in all products that satisfy the input feature expression, without going through any of the excluded states. This modified reachability algorithm returns a featured set of states: a mapping of states to feature expressions representing the set of states reachable under a given product. This represents a featured SCC in which the set of states that are members of the SCC depends on the software product.

Algorithm 6 starts by initializing the featured SCC that will be computed, denoted *FSCC*, to comprise only the input state and the input feature expression (line 8). It then proceeds to compute, for the products that satisfy the input feature expression (λ_{start}), (1) which states are reachable from the input state in those products, and (2) in which subset of those products are those states reachable.

To compute this modified reachability relationship, the algorithm uses the stack *StatesFExpToExplore*, comprising tuples of <state, feature expression>, to track the states that it must explore and for which products it must explore them. The algorithm initializes the stack *StatesFExpToExplore* by pushing the tuple of the input state and input feature expression $< s_{start}, \lambda_{start} >$ onto it (lines 9-10).

The algorithm then enters a loop where it processes the elements of this stack (lines 11-25). The algorithm peeks at the top element $< s, px >$ of the stack and iterates through each transition that starts at state s, and whose corresponding feature expression is satisfiable in conjunction with px (lines 14-18) and checks whether exploring $< s_{dst}, px \wedge \gamma(t) >$ could result in an update (expansion) to the reachability relationship represented by $FSCC$, specifically to $FSCC(s_{dst})$. Because s_{dst} in products that have already been added

Alg. 6: Reachability computation for the transpose of an FTS, excluding states already assigned to an SCC.

```
1   Procedure VisitDFS-For-SCC(fts, s_start, λ_start, InPrevSCC)
2   Input: fts = ( S, I, A, T, d, γ): featured transition system
3   Input: s_start: initial state of the SCC
4   Input: λ_start: feature expression associated with state s_start in the SCC
5   Input: InPrevSCC : S → 𝔹(N): the (featured) set of states that have
        already been assigned to an SCC
6   Output: FSCC : S → 𝔹(N): A featured SCC.
7   begin
8       FSCC ← {(s_start, λ_start)}
9       StatesFExpToExplore ← Empty stack of tuples of < state,
                                                    feature expression >
10      StatesFExpToExplore.push((s_start, λ_start))
11      while (¬StatesFExpToExplore.isEmpty()) do
12          < s, px > ← Stack.peek()
13          new ← Empty set of tuples of < state, feature expression >
14          for each t = (s, α, s_dst) ∈ T such that px ∧ γ(t) is satisfiable
15            newFExp ← px ∧ γ(t)
16            if newFexp ∧¬ InPrevSCC(s_dst) ∧¬ FSCC(s_dst) is satisfiable then
17                new.add(< s_dst, newFexp>)
18            end-if
19          if new = ∅ then
20            pop(StatesFExpToExplore);
21          else
22            Choose <sNext, pxNext> ∈ new
23            FSCC(sNext) ← FSCC(sNext) ∨ (pxNext ∧ ¬ InPrevSCC(sNext))
24            StatesFExpToExplore.push(<sNext, pxNext ∧ ¬ InPrevSCC(s)>)
25          end-if
26      return FSCC
27  end
```

to other SCCs (i.e., products in $[\![InPrevSCC(s_{dst})]\!]$) or that already belong to the SCC being computed (i.e., products in $[\![FSCC(s_{dst})]\!]$) should not be (re) added to the SCC being computed, the algorithm first checks whether $px \wedge \gamma(t) \wedge \neg InPrevSCC(s_{dst}) \wedge \neg FSCC(s_{dst})$ is satisfiable (lines 15-16). If so, it adds the tuple $< s_{dst}, px \wedge \gamma(t) >$ to the set *new* (line 17).

After processing all those transitions, if the set *new* is empty – that is, there are no new elements to explore from the current state – then the algorithm pops the top element of the stack (line 20). Otherwise it takes a state and feature expression that is a new element, updates $FSCC$ with it, and pushes the new element onto the stack (lines 22-24). It then continues processing elements of the stack until no more remain and finally returns $FSCC$.

4.3.3 Maximum Average-Weight Cycle Computation

Finally, for the SCCs identified in Algorithm 6, we need to determine their maximum average-weight cycles. We adapt Karp's original algorithm [68], which determines the maximum average-weight cycle in an SCC, to be feature-aware. We show the adapted algorithm in Algorithm 7.

In Karp's original algorithm, an arbitrary "source state" u is chosen. The algorithm uses a function F that records for each state v and for each possible path length $k \in \{0 \ldots |SCC|\}$ the maximum weight of a path of length exactly k from the "source state" u to state v. This function F is computed in an iterative manner, and is then used to determine the average-weight of the maximum average-weight cycle.

We adapt Karp's algorithm to compute the average-weight of the maximum average-weight cycle of a *featured* SCC of a *featured* transition system. Therefore, our approach needs to adapt the algorithm to output a different average weight for each set of products. Specifically, our algorithm inputs a features a featured SCC and outputs a set C of tuples $< \lambda, w >$, which records that the products that satisfy the feature expression λ in the featured SCC have a maximum average-weight cycle with a average weight of w.

In the following paragraphs, we describe our adapted algorithm. Note that the maximum weight of a path of length exactly k from the "source state" to a state v will vary for each product or set of products. Thus, our approach adapts function F to be a relation, denoted *WeightMWP* ("MWP" is an abbreviation for Maximum Weight Path), that comprises tuples $< k, v, \lambda, w >$, where feature expression λ represents a set of products, and w is the maximum weight of a path from the "source state" s_{start} to state v of length exactly k.

Alg. 7: Computation of the maximum average-weight cycle in a featured SCC.

Procedure Average-Cycle-SCC(wfts, FSCC, s_{start})
Input: wfts = (S, I, A, T, d, γ, W): a weighted featured transition system
Input: FSCC: $S \to \mathbb{B}(N)$: a featured SCC
Input: s_{start}: initial state of the featured SCC
Output: C: $\mathbb{B}(N) \to \mathbb{R}$: a featured average weight of the maximum average-weight cycle
begin
  FSCCStates = $\{s | s \in S, FSCC(s) \not\models \bot\}$
  WeightMWP $\leftarrow \emptyset$
  for $k = 0, \ldots, |FSCCStates|$ and $s \in$ FSCCStates$\backslash\{s_{start}\}$
    WeightMWP.Insert($< k$, s, FSCC(s), $-\infty >$)
  WeightMWP.Insert(<0, s_{start}, FSCC(s_{start}), $0 >$)
  for $k = 1, \ldots, |$FSCCStates$|$
    for each $s \in$ FSCCStates
      for each $t = (s_{t_{src}}, \alpha$,s$) \in T$ such that $FSCC(s_{t_{src}}) \not\models \bot$
        $\lambda_1 = \gamma(t)$
        for each $\lambda_2, w_{\lambda_2}, \lambda_3, w_{\lambda_3}$ such that $< k, s, \lambda_2, w_{\lambda_2} > \in$ WeightMWP and $< k-1, s_{t_{src}}, \lambda_3, w_{\lambda_3} > \in$ WeightMWP
          if $\lambda_1 \wedge \lambda_2 \wedge \lambda_3 \not\models \bot$ and $w_{\lambda_3} + W(t) > w_{\lambda_2}$ **then**
            WeightMWP.insert($< k$, $s, \lambda_2 \wedge \lambda_3 \wedge \lambda_1, w_{\lambda_3} + W(t) >$)
            WeightMWP.insert($< k$, $s, \lambda_2 \wedge \neg(\lambda_3 \wedge \lambda_1), w_{\lambda_2} >$)
            WeightMWP.remove($< k$, $s, \lambda_2, w_{\lambda_2} >$)
          end-if
  M $\leftarrow \emptyset$
  C $\leftarrow \{< FSCC(s_{start}), -\infty >\}$
  for each $s \in$ FSCCStates
    M.insert($< s, FSCC(s), +\infty >$)
    for $k = 0, \ldots, |$FSCCStates$| - 1$
      for each $\lambda_1, w_1, \lambda_2, w_2, \lambda_3, w_3$ such that $< s, \lambda_1, w_1 > \in M$ and $< |$FSCCStates$|, s, \lambda_2, w_2 > \in$ WeightMWP and $< k$, $s, \lambda_3, w_3 > \in$ WeightMWP
        if $\lambda_1 \wedge \lambda_2 \wedge \lambda_3 \not\models \bot$ and $w_1 > (w_2 - w_3)/(|$FSCCStates$| - k)$ **then**
          M.insert($< s, \lambda_1 \wedge \lambda_2 \wedge \lambda_3, (w_2 - w_3)/(|$FSCCStates$| - k) >$)
          M.insert($< s, \lambda_1 \wedge \neg(\lambda_2 \wedge \lambda_3), w_1 >$)
          M.remove($< s, \lambda_1, w_1 >$)
        end-if
    for each $\lambda_1, w_1, \lambda_2, w_2$ such that $< \lambda_1, w_1 > \in C$ and $< s, \lambda_2, w_2 > \in M$
      if $\lambda_1 \wedge \lambda_2 \not\models \bot$ and $w_1 < w_2$
        C.insert($< \lambda_1 \wedge \lambda_2, w_2 >$)
        C.insert($< \lambda_1 \wedge \neg\lambda_2, w_1 >$)
        C.remove($< \lambda_1, w_1 >$)
      end-if
  return C
end

First, our algorithm initializes relation WeightMWP, such that for each state s (except source state s_{start}) and for all the products that satisfy feature expression $FSCC(s)$ (for which state s is part of the SCC), the maximum weight of a path of length exactly k (for k ranging from 0 to $|FSCCStates|$) from the "source state" s_{start} to state s is set to $-\infty$ (lines 9-10). Unlike Karp's algorithm, which chooses an arbitrary "source state", we choose the "initial state" of the SCC, denoted s_{start}, as the "source state". Because the set of products in which s_{start} is part of the SCC will be a superset of the set of products for which any other state is part of the SCC, this ensures that s_{start} will be present in all products that are part of the SCC. To complete the initialization of relation WeightMWP, our algorithm inserts tuple <0, s_{start}, FSCC(s_{start}), $0>$ to represent that the state s_{start} can be reached from itself through a path of length and weight 0 (the empty path) for all products in which state s_{start} is part of the featured SCC (line 11).

Next, our algorithm iterates over all transitions of the weighted featured transition system to refine relation WeightMWP. For each transition t and each value of k, the algorithm updates the weight associated with the triplet $< k, s, \lambda >$ in relation WeightMWP if a higher-cost path from the "source state" exists – that is, if the sum of the cost of executing transition t and the weight of a $k-1$ length path from s_{start} to transition t's source state $s_{t_{src}}$ is *higher* than the weight that is currently associated with triplet $< k, s, \lambda >$ (line 16-20). Note that for products in which transition t does not exist, the weight associated with the tuple stays the same. This concludes the description of the calculation of relation WeightMWP.

To explain the next two loops of the algorithm, we recall Karp's theorem that is presented as Formula 4.1 in this chapter's introduction, but to align the notation with the algorithm we rename certain variables:

$$\max_{s} \min_{k<|SCC|} \frac{WeightMWP[|SCC|, s] - WeightMWP[k, s]}{|SCC| - k} \tag{4.8}$$

- Karp shows that in any SCC there exists a state $s*$ that is part of the maximum average-weight cycle in the SCC such that

  $$min_{k<|SCC|} \frac{WeightMWP[|SCC|, s*] - WeightMWP[k, s*]}{|SCC| - k}$$

  equals the average weight of steps in the maximum average-weight cycle. Figure 4.5 shows the paths that are associated with state $s*$: the path shown in red is the path of length $|SCC|$ from s_{start} to $s*$ whose total weight is $WeightMWP[|SCC|, s*]$; the path shown in green is the path of length k (for the minimizing k) from s_{start} to $s*$

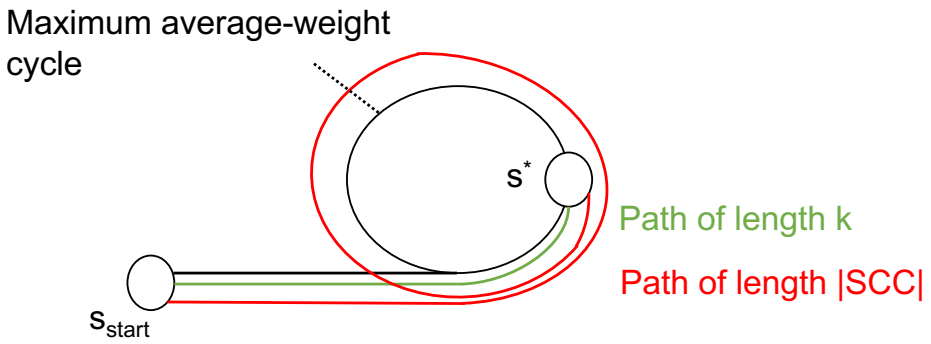


Figure 4.5: A depiction of two paths that are associated with Formula 4.8 and state s*: the red path is the maximum weight path of length $|SCC|$ from source state s_{start} to state s* and the green path is the maximum weight path of length k from source state s_{start} to state s* (for the minimizing k). The maximum average-weight cycle is the cycle in black and has length $|SCC| - k$. The average weight of steps in the maximum average-weight cycle is given by the weight of the red path, WeightMWP($|SCC|, s*$), minus the weight of the green path, WeigthMWP($k, s*$), divided by the length of the maximum average-weight cycle, $|SCC| - k$.

whose total weight is $WeightMWP[k, s*]$. The maximum average-weight cycle will be of length $|SCC|-k$, where k is the minimizing k. The total weight of the maximum average-weight cycle will be equal to the weight of the red path minus the weight of the green path. The average weight of steps in the maximum average-weight cycle will be the total weight divided by $|SCC| - k$.

- For any state, s_{other}, the expression

$$\min_{k<|SCC|} \frac{WeightMWP[|SCC|, s_{other}] - WeightMWP[k, s_{other}]}{|SCC| - k}$$

  is less than or equal to the average weight of steps in the maximum average-weight cycle because:

  - If there does not exist a path of length $|SCC|$ from s_{start} to s_{other}, then the expression $WeightMWP[|SCC|, s_{other}]$ will equal $-\infty$. Consequently, $\min_{k<|SCC|}[(WeightMWP[|SCC|, s_{other}] - WeightMWP[k, s_{other}])/(|SCC| - k)]$ will be less than the average weight of steps in the maximum average-weight cycle.

– If there does exist a path of length $|SCC|$ from s_{start} to s_{other}, then this path must include a cycle, which we denote as c_{other}.

  By removing the cycle c_{other} from such path, we obtain a path of length $(|SCC| - |c_{other}|)$ from s_{start} to s_{other} whose total weight is $WeightMWP[|SCC|, s_{other}]$ minus the total weight of cycle c_{other}.

  Because $WeightMWP[|SCC| - |c_{other}|, s_{other}]$ is equal to the maximum weight of a path from s_{start} to s_{other} of length $(|SCC| - |c_{other}|)$, then $WeightMWP[|SCC| - |c_{other}|, s_{other}]$ is greater than or equal to $WeightMWP[|SCC|, s_{other}]$ minus the total weight of cycle c_{other}.

  So, $WeightMWP[|SCC|, s_{other}] - WeightMWP[|SCC| - |c_{other}|, s_{other}]$ is less than or equal to the total weight of cycle c_{other}.

  By dividing such expression by the length of the cycle c_{other} (i.e.,$|c_{other}|$), we obtain that

  $$(WeightMWP[|SCC|, s_{other}] - WeightMWP[|SCC| - |c_{other}|, s_{other}])/(|c_{other}|)$$

  is less than or equal to the average weight of steps in cycle c_{other}.

  Because $\min_{k<|SCC|} \ldots$ takes a minimum across all possible values of k, and the aforementioned expression can be obtained by replacing k with $|SCC| - |c_{other}|$, then $\min_{k<|SCC|} \ldots$ is less than or equal to the average weight of steps in cycle c_{other}.

  Because the average weight of steps in cycle c_{other} is less than or equal to the average weight of steps in the maximum average-weight cycle,

  $$\min_{k<|SCC|}[(WeightMWP[|SCC|, s_{other}] - WeightMWP[k, s_{other}])/(|SCC| - k)]$$

  will be less than or equal to the average weight of steps in the maximum average-weight cycle.

Because Formula 4.8 calculates the maximum value of $\min_{k<|SCC|}[WeightMWP \ldots]$ across all states s in the SCC, this implies that the value of Formula 4.8 equals the average-weight of steps in the maximum average-weight cycle of an SCC.

Returning to our algorithm (line 22), the next step is to determine for each state $s_{candidate}$ whether it can play the role of $s*$: that is, whether there exists a maximum-weight path from s_{start} to $s_{candidate}$ of length $|FSCCStates|$, and whether there exists a maximum average-weight cycle from $s_{candidate}$ to $s_{candidate}$ in the tail of such a path. As we previously explained, the expression

$$min_{k<|SCC|}[(WeightMWP[|SCC|, s] - WeightMWP[k, s])/(|SCC| - k)]$$

equals the average weight of steps in the maximum average-weight cycle for state $s = s*$, and is less than or equal to the average weight of steps in the maximum average-weight cycle for all other states. Because the value of this expression will vary for different sets of products, our algorithm records these weights in a relation, denoted M, that comprises tuples $< s, \lambda, w >$, where for state s in all products satisfying feature expression λ, the value of the aforementioned expression is w.

To compute relation M, the algorithm first initializes M to $+\infty$ for all states in the SCC and all products for which state s is part of the SCC (line 25). For each possible value of k (between zero and $|FSCCStates| - 1$), the algorithm checks if there are any products λ for which $(WeightMWP(|FSCCStates|, s) - WeightMWP(k, s))/(|FSCCStates| - k)$ is less than the weight w currently stored as tuple $< s, \lambda, w >$ in M, and if so the algorithm updates M (lines 28-32).

The last loop of the algorithm identifies the state $s*$ that maximizes the expression

$$min_{k<|FSCCStates|} \frac{(WeightMWP(|FSCCStates|, s) - WeightMWP(k, s))}{(|FSCCStates| - k)}$$

thereby identifying the average weight of steps in the maximum average-weight cycle. Because this value varies for different products, the algorithm uses a relation, denoted C, of tuples $< \lambda, w >$, where w is the maximum value of $min_{k<|FSCCStates|} \ldots$ across all states in all products that satisfy feature expression λ.

To compute relation C, the algorithm first initializes C to $-\infty$ for all products (line 23). Using the average weights of cycles in relation M, the algorithm checks if there are any products and states s in M that have a cycle with a higher average weight than the value associated with those products in C. If so, it updates the value associated with those products in C (lines 34-37), so that C records the "maximum" value of M across all states for each set of products. Once the algorithm has iterated over all states in the featured SCC, the calculation of C is complete and the algorithm returns C (line 39).

4.4 Implementation and Evaluation

We have implemented our algorithms within ProVeLines (see subsection 2.4.3 for a brief overview of ProVeLines), a product line of verifiers for SPLs [30]. ProVeLines takes as input a specification written in fPromela (see subsection 2.4.2 for a brief overview of fPromela), a feature-aware extension of the Promela language [62] (see subsection 2.4.1 for a brief overview of Promela), which we have extended further to specify transition weights.

We have modified the code of ProVeLines to include weights on transitions and to perform family-based and product-based computations of the maximum average-cycle algorithm.

As an example, Fig. 4.6 shows part of our extended fPromela specification of the taxi-shuttle example from Section 4.2. The three features Shuttle, Taxi and License are declared at the beginning of the file (lines 1-6). The main process is called "taxi" as indicated by the line "active proctype taxi() {" (lines 9-22). Inside such process, we show a transition from AirportR (Airport dropoff location) to AirportP (Airport pick up location) annotated with a weight of -5 (lines 11-12). Another transition in the shown snippet is from Pick-Up location one directly to Pick-Up location two (lines 13-18); this transition is guarded by feature Shuttle and annotated with a weight of 15.

As we have previously mentioned, our algorithm determines whether a formula is satisfiable by performing a call to satisfiability solver *MiniSat* [37], which has a run time complexity of $O(2^N)$ where N is the the number of variables in the formula. We recall that our algorithms use feature expressions to represent sets of configurations. Our implementation of the algorithms use *Ordered Binary Decision Diagrams* (BDDs) [12] to encode feature expressions. More concretely, our implementation of the algorithms use the *Colorado University Decision Diagram* (CUDD) [100] library to implement and manipulate BDDs that represent feature expressions. We also note that our family-based maximum limit-average algorithm does not improve the worst-case computational complexity compared to individually computing the maximum limit-average for each software product.

4.4.1 Subject Systems

For testing and experiments, we used a generalization of the taxi-shuttle example from Section 4.2 in which the number of extra licenses is parameterized. This variant has N different extra-license features $L_1, \ldots, L_N$, each with their own Pickup-ext$_i$ and Release-ext$_i$ states and transitions that are guarded by the feature L_i. For our experiments, N ranged from one to eleven.

As a second case study, we created a WFTS representing a mine pump controller, based on an example in [28]. The original example models a system that controls a water pump to balance the levels of water and methane in a mine shaft. The water level needs to be low for operation; and high levels of methane can lead to an explosion, and thus need to be avoided. The system consists of a water pump, a methane sensor, and a command module that activates or deactivates the water pump. It is modeled as the parallel composition of five individual FTSs, which model the command module, the user (who sends start/stop commands), the methane alarm, the water sensor, and the methane sensor. The original

```
1  typedef features {
2      bool Shuttle;
3      bool Taxi;
4      bool License
5  };
6  features f;
7  int current = 0;
8  ...
9  active proctype taxi() {
10     do ::
11         if  ::  (current == AIRPORTR);
12                 current = AIRPORTP [-5];
13             ::  (current == PICKUP1);
14                 if  ::  f.Shuttle;
15                         current = PICKUP2 [15];
16                     ::  !f.Shuttle;
17                         skip;
18                 fi;
19             ...
20         fi;
21     od;
22 }
```

Figure 4.6: Part of the fPromela specification of the taxi-shuttle example

Table 4.1: Maximum limit-average values for the taxi example. Pickup-N is abbreviated as PN, Release-N is abbreviated as RN, airport is abbreviated as AP, extended pickup location is abbreviated as PE, and extended release location is abbreviated as RE.

Product	Maximum Limit-Average	Cycle
$\emptyset$	12.17	AP→R2→P2→AR→AP
$\{L\}$	12.17	AP→R2→P2→AR→AP
$\{S\}$	12.88	AP→R2→R1→P1→P2→AR→AP
$\{T\}$	14.00	P1→R2→P2→R1→P1
$\{L, S\}$	13.30	AP→R2→R1→RE →PE→P1→P2→AR→AP
$\{L, T\}$	14.00	P1→R2→P2→R1→P1
$\{S, T\}$	14.33	P1→P2→R1→P1
$\{L, S, T\}$	14.60	PE→P1→P2→R1→RE→PE

mine pump controller example had nine features [28]. We adapted the original version of the mine pump controller example to use only two features as additional features caused our analysis to take too much time (over 24 hours). Our WFTS of the mine pump controller has two optional features (the command module and the methane sensor) and four products; and we annotated transitions in the main module with artificial weights. The ProveLines encoding of the mine pump controller example has 9441 different states.

4.4.2 Results

In Table 4.1, we report the complete results of the maximum limit average for all products of the extended taxi example from Section 4.2. The first column describes each product, the second column states the maximum limit-average value for each product, and the third column lists an example cycle in each product with the given maximum limit-average value (Pickup-N is abbreviated as PN, Release-N is abbreviated as RN, airport is abbreviated as AP, extended pickup location is abbreviated as PE, and extended release location is abbreviated as RE).

Table 4.2 shows the running times for both family-based and product-based maximum limit average analyses of both the parameterized taxi example and the mine pump controller example. We ran both the family-based and product-based analyses ten times each and show the average times and standard deviations. Table 4.2 also shows the average speedup obtained by the family-based approach analysis versus the product-based analysis. We executed both analyses on a Macbook Pro with 8 GB of RAM, and with an 2.7 GHz Intel

Table 4.2: Average time taken by the family-based and product-based maximum limit-average computation for the taxi and the mine pump controller examples

.

# features	# products	# states	family-based (seconds)[a]	product-based (seconds)[a]	speedup
Parameterized taxi example					
3	8	52	0.25 ± 4.57 %	0.27 ± 9.44 %	1.08
4	16	75	0.30 ± 3.47 %	0.56 ± 1.64 %	1.87
5	32	98	0.44 ± 2.99 %	1.04 ± 9.04 %	2.36
6	64	121	0.80 ± 4.15 %	2.19 ± 2.19 %	2.74
7	128	144	1.83 ± 13.2 %	4.89 ± 1.36 %	2.67
8	256	167	3.86 ± 1.07 %	10.6 ± 2.01 %	2.75
9	512	190	10.8 ± 8.95 %	23.3 ± 2.10 %	2.16
10	1024	213	24.6 ± 6.26 %	51.0 ± 1.94 %	2.07
11	2048	236	63.3 ± 5.05 %	115 ± 1.79 %	1.82
12	4096	259	142 ± 5.27 %	252 ± 1.47 %	1.77
13	8192	282	308 ± 1.55 %	554 ± 1.33 %	1.80
Mine pump controller example					
2	4	9441	292 ± 2.79 %	111 ± 7.61 %	0.38

[a]Mean ± Stdandard Deviation

Core i5 processor.

For the parameterized taxi example, our family-based approach is approximately twice as fast as the product-based approach, albeit the difference in speed decreases as the number of features increases beyond ten. Our family-based analysis is not much faster than the product-based analysis in the parameterized taxi examples with fewer than five features. This could be due to the overhead required to execute our family-based analysis (e.g., setting up the data structures). For the mine pump controller, however, the family-based analysis takes more than twice as long as the product-based analysis.

4.4.3 Discussion

Our results for the parameterized taxi example are as expected. In the taxi example, featured SCCs are shared across different software products, so that a single computation over a featured SCC provides results that can be re-used across multiple products. This can explain why the required time is reduced when using a family based-approach versus a product-based approach.

The mine pump controller example has very few products, and SCCs are not shared across products. Hence the family-based approach fails to "lump" products into families and is, thus, slower than the product-based analysis. Additionally, this example has a much larger state space than the taxi example, hence both product-based and family-based analysis take much longer to execute when applied to the mine pump example than when applied to the taxi example.

By profiling our implementation and measuring the execution times of the various steps involved in our approach, we found that computing the maximum average-weight cycle for featured SCCs takes most of the time in the family-based approach. As mentioned in subsection 4.1.1, the algorithm to compute the maximum average-weight cycle takes time $O(V \times E)$, so it is not surprising that executing this algorithm takes most of the time.

Most importantly, we found that for our mine-pump controller example, different products induce different sets of finishing times, and that there is very little sharing of finishing times across products of featured SCCs. Therefore, the family-based approach does not improve the performance for this example, and the overhead introduced by the family-based approach means it is substantially slower than the product-based analysis.

This suggests that a key aspect that determines whether the family-based analysis will be faster than a product-based approach is whether multiple products in the SPL will induce the same set of finishing times (and thus share similar strongly connected components). As further evidence of this hypothesis, in the taxi example, all products shared

the same set of strongly connected components and the family-based analysis performed much better in it. As such, it is possible that the results of the featured finishing-times tree could be used to decide whether to proceed with a family-based analysis or a product-based analysis. This warrants further investigation.

4.5 Related Work

Product-Line Analysis Lauenroth et al. [73] 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 the form $EG\ f_1$ 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 the maximum average-cost cycle for each product or set of products. Hence our family-based analysis computes (featured) strongly connected components whereas Lauenroth et al.'s tool only searches for reachable cycles to perform CTL model checking. Their approach does not analyze models that include weights. They do not compare the performance of their family-based approach with respect to a product-based approach.

Classen et al. [26] adapt the standard algorithm for model checking transition systems with respect to properties expressed in linear temporal logic (LTL) to enable analysis of a product line represented as a featured transition system. Their family-based model checker that analyzes a product-line model is between 2 and 38 times faster than analyzing individually the corresponding model of each product. Although they represent products symbolically, they still represent the featured transition system using explicit states and transitions. In subsequent work, Classen et al. [24] extend their approach to featured 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 faster than verifying each product individually. More recently, Ben-David et al. [6] have adapted SAT-based model checking of safety properties to a family-based approach and showed that their model-checker is substantially faster than the methods developed by Classen et al.

In follow-up work, Cordy et al.[32] show how to perform family-based model checking of

software product lines that have been extended with: i) numeric feature attributes and ii) multi-features – the ability to have multiple instances of a feature. In their work, transitions are annotated with expressions that range over features, feature attributes, and feature cardinalities instead of being annotated with only with feature expressions (i.e., boolean formulas ranging over features). Because of the need to support non-boolean formulas, they use Satisfiability Modulo Theory (SMT) solver Z3 [35] to implement their family-based SPL model checking algorithm. As an alternative, they implement their family-based SPL model checking algorithm by using a pre-processing step that converts non-boolean formulas into boolean formulas by introducing intermediate variables and associated boolean constraints to represent numeric constraints. They compare the execution time of this two methods in model-checking five different properties of a satellite communication module case study and conclude that the pre-processing method is between 92 and 171 times faster than the Z3-based method. They then compare the execution time of their pre-processing family-based model checking method against the execution time of analyzing each product individually for one case study – a model of the sieve of Erasothenes that checks for primes in numbers between 1 and n, with n ranging from 30 to 48. Their results are inconclusive – for values of n around 30 both methods take approximately the same time, whereas as n increases to 50 the family-based model checking algorithm is up to two times faster than analyzing each product individually. Unlike our approach, they do not associate numeric values to the execution of transitions.

Additionally, Cordy et al. [31] present a method to perform family-based analysis of real-time software product lines. They introduce *Featured Timed Automata* that is an extension of *Timed Automata*, which is the standard formalism that is used to model and verify real-time systems with continuous time. Featured Timed Automata support modelling the different software products of an SPL in a single timed model. They implement a family-based model checking algorithm for real-time software product lines, that are expressed as a featured timed automata, for a subset of *Timed Computational Tree Logic* (TCTL). They compare the execution time of their family-based model checking method against the execution time of model checking each software product individually for a single case study and obtain mixed results: their family-based approach is 2.0 times slower than analyzing each product individually for an instance of the case study with 6 features, whereas it is 2.1 and 5.5 times faster than analyzing each product individually for instances of the case study with 10 and 13 features respectively.

Limit-Average Cost Quantitative methods are important in performance analysis [63], reliability analysis [97], and other areas of software engineering. Long-term average values (i.e, maximum/minimum limit average) are often used, for example, to measure mean

time between failures or average power consumption [55, 58]. Karp's [68] algorithm is the standard way to compute the maximum (or minimum) limit-average cost of executing a weighted state-transition model. The maximum limit-average cost ignores one-time costs that might have minimal impact over time in a continually executing system, so it can accurately represent the long-term average cost of system execution for those type of systems. We are not aware of any family-based analysis methods which compute the limit-average cost for all products in a software product line.

4.6 Conclusions and Future Work

We have introduced a family-based algorithm to compute the maximum limit average of quality attributes in a software product line. Our algorithm is based on featured extensions of the standard algorithms for computing maximum limit average and is able to compute the maximum limit average of quality attributes for all products in one run.

We have implemented our algorithm by extending ProVeLines, an existing tool for model checking software product lines, to include capabilities to compute the maximum limit average for product line models annotated with weights on transitions. We have used our implementation to evaluate our approach by comparing the performance of our algorithm against a product-based (enumerative) approach.

We have shown that our family-based approach speeds up analysis compared to a product-based analysis for one SPL that has a large number of products and whose products share the same strongly connected components; our family-based approach increases the speed of analysis by a factor of two. For the other SPL that we analyzed, our family-based approach is slower than a product-based approach. This SPL has more states, a more complex behaviour, and fewer products, which is known to be less cost-effective to analyze using family-based techniques. Moreover each product tends to have different strongly connected components, which limits the extent to which our family-based approach can reuse partial results across different software products. We think that our family-based approach will perform best for SPLs in which many products share the same set of strongly connected components. Furthermore, the results of the featured finishing-times tree (i.e., how much sharing of finishing-times there is across different products) could potentially be used to determine, at an early stage in the analysis of the WFTS, whether a family-based or product-base computation of the SPL's maximum limit average will be more efficient.

Chapter 5

Learning Timed Featured Transition Systems

As we discussed in Chapter 2, a timed (or weighted) transition system model extends a state-transition system model with weights on each transition, which can represent the time taken (e.g., [78]) or the consumption of a resource (cost) due to the transition's execution. Model checkers can then analyze a timed state-transition system model to determine whether a software system will satisfy it's timing requirements. However, obtaining accurate timing models is a key barrier to the efficacy of timed model checking and is even more challenging for a software product line because a transition's execution time can vary among different software products. One can manually annotate an untimed transition model with manually derived estimates of transitions' costs, but such estimates are suspect. There exist automated approaches that extract an untimed behaviour model from a system's execution traces [8, 75, 77]; and Perfume [86] and Timed K-Tail [92] adapt these approaches ([8] and [77]) to extract a timed transition system and a timed automaton, respectively. Both Perfume and Timed K-Tail simultaneously learn a model of a system's behaviour as well as the timing performance of the system — whereas in some cases a behaviour model might be available and we would be interested in learning only the timing attributes.

Currently, state-of-the-art methods learn a Performance-Influence model [98] for a software product line that estimates a product's execution time based on the static contributions of the product's features and feature combinations to the product's execution time. The Performance-Influence model does not model the internal computation steps of the SPL, but instead estimates how long would each software product takes to execute a certain task – such as encoding a set of videos, or compressing a set of files – solely based

on which features are present in a software product. We hypothesize that obtaining a more fine-grained model that estimates how long each transition will take to execute in each software product can lead to models that permit more precise analysis (such as model checking of timing requirements of a software product line). In this chapter, we present an approach to learn an accurate timed state-transition model for a software product line and an evaluation of our hypothesis. The presence or absence of a feature in a software product can impact whether a transition will be available in a product and how long it will take the transition to execute. For example, in the Unmanned Air Vehicle (UAV) SPL, a transition that causes the UAV to follow a certain target might be available only in the products that include both the feature Computer Vision (CV) and the feature Global Positioning Systems (GPS). Thus, a timed state-transition model for an SPL must include a function that maps a software product and a transition to how long it would take to execute that transition in that product.

In this chapter, we propose an approach that learns the weights for an SPL's weighted behavioural model (i.e., a weighted featured transition system model) from a set of execution traces from a sample of the software products. Specifically, we apply supervised learning techniques (Linear Regression and Regularized Linear Regression) to learn, for each transition, a regression function that estimates how much time the transition will take to execute in each software product that includes it. We assessed the accuracy of our approach using two subject systems: X264, an open source video encoding system, and Autonomoose, a self-driving system. We compared our approach against two different state-of-the-art methods: a Performance-Influence model [98] and Perfume [86].

The result is a timed transition model whose predicted weight on each transition in each product has a mean error that ranges from 3.8% to 193.0% when applied to the subject systems. However, our method's estimates of the *overall execution times* of an SPL product executing an input task, have an accuracy that is similar to that obtained by the existing Performance-Influence model [98] whose estimates are based on the features (and feature combinations) that are present in that product.

Our contributions are:

- We present and evaluate for the first time a method to learn timed behaviour models for software product lines.
- We evaluate the quality of the learnt timed models of an SPL.
- We compare the accuracy of our method's learnt models to two different the state-of-the-art methods: a Performance-Influence model [98], which estimates the execution

time of an SPL product on an input task, and Perfume [86], which is a tool that learns a timed behaviour model from timed execution traces of a single software system.

The rest of the Chapter is organized as follows. In Section 5.1 we describe our approach, in Section 5.2 we describe our evaluation methodology and we discuss our empirical results, in Section 5.3 we discuss related work, and in Section 5.4 we present our conclusions.

5.1 Approach

Our objective is to learn, for each transition t in a timed state-transition model, a function that predicts the transition's execution time in each software product of a product line. We choose to analyze each transition independently of other transitions – that is, we create $|T|$ distinct learning problems where $|T|$ is the number of transitions in the featured transition system. Our approach takes as input a featured transition system (FTS), which is an untimed behaviour model of an SPL, and a set of timed execution traces of a sample of software products. Our approach produces $|T|$ compact functions, where each function predicts the execution time of one transition in every software product in which that transition is available. In the rest of this section, we describe the learning of the function for an individual transition $t \in T$.

We set up our learning task as a *regression* task. Regression is a type of supervised learning in which the objective is to learn a function, denoted f, that best fits a set of observed input/output pairs $\{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \ldots\}$, where each pair consists of a vector of input-variable values and a single numerical output value. Ideally, when the learnt function f is subsequently applied to the input of an unseen input/output pair, the function's output matches the pair's output (e.g., ideally $f(\vec{x}) = y$ for all pairs). The set of input/output pairs used to learn the function f is called a *training set*. A separate set of input/output pairs, called a *testing set*, is used to evaluate the accuracy of the learnt function.

We need to convert the input to our approach (a set of timed execution traces) into the input expected by the regression task for transition t (a set of input/output pairs). First, we extract from the traces all execution steps that involve the execution of transition t. We emphasize that each transition t can execute multiple times in an execution trace and each one of its executions can take a different amount of time. For each extracted execution step, we generate an input/output pair that comprises the software product that executed that step, represented as a characteristic function on the set of features, and the duration of that execution step. Specifically, each input consists of a vector of $|F|$ boolean variables $\vec{x} = [x_1, x_2 \ldots x_{|F|}]$ where the i^{th} input variable denotes whether the i^{th} feature

of the SPL is present in the product. For example in the Unmanned Air Vehicle SPL, the input/output pair ($\vec{x} = [x_1 = 1, x_2 = 0, x_3 = 0], y = 10$) would represent an execution of transition t in a product consisting exclusively of feature GPS (i.e., feature "1"), where that execution of transition t took 10 time units to complete. We randomly partition these input/output pairs into a training set and a testing set.

In the following two subsections we describe how we apply *Linear Regression* and *Regularized Linear Regression* [103, 104] to learn a function f_t that estimates more generally the execution time of a transition t in each product of an SPL.

5.1.1 Linear Regression

Linear Regression [9] is a widely used regression method in which the learnt function is linear in its input variables. The learnt function is called a *linear regression function*: it relates a weighted sum of the input variables plus a fixed constant to an estimated value of the output variable. In our case, the weights $w_1, \ldots, w_{|F|}$ represent estimates of each feature's contribution to transition t's execution time, and the fixed constant w_0 represents the baseline time that transition t takes to execute. Thus, the learnt function f_t estimates transition t's execution time in each product of a software product line with $|F|$ features; it has the following form:

$$f_t(\vec{x} = [x_1, x_2 \ldots x_{|F|}]) = w_0 + \sum_{i=1}^{|F|} w_i \times x_i$$

The objective is then to find the "best" vector of weights $\vec{w} = w_0, w_1, \ldots, w_{|F|}$ such that the linear regression function f_t provides the best prediction of t's execution time in each software product of the SPL. To quantify how well a linear regression function, f_t, fits a training set $TrainingSet = \{(\vec{x}_1, y_1)(\vec{x}_2, y_2) \ldots\}$ of pairs of software products and one of the execution times of transition t for that product, our approach uses a *loss function* that measures the *squared difference* between the function's prediction of t's execution time $f_t(x_i)$ and t's observed execution time y_i:

$$\sum_{(\vec{x},y) \in TrainingSet} (f_t(\vec{x}) - y)^2$$

Our approach uses the *ordinary least squares* (OLS) method to identify the linear regression function whose corresponding weight vector minimizes the value of the loss function on the training set. This method identifies the weight vector by solving a set of matrix equations

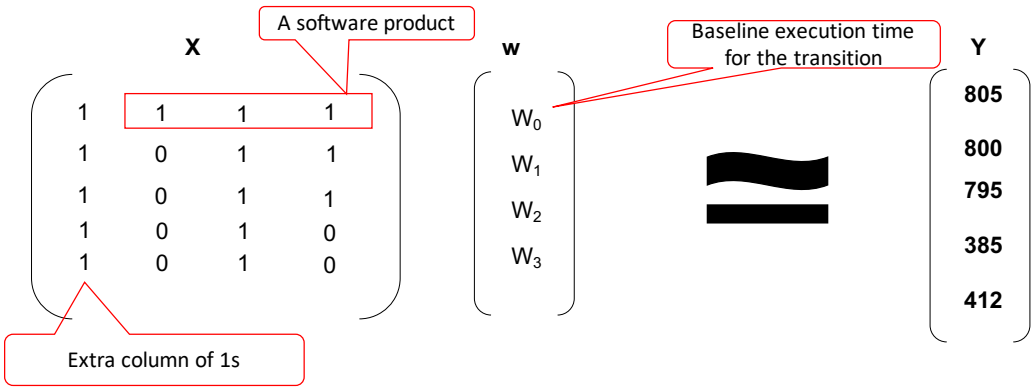


Figure 5.1: An example linear regression task: estimating transition t's execution time for the different software products of the Unmanned Air Vehicle SPL (introduced in Chapter 2). Each labelled example in the training set comprises: the software product of one of transition t's executions from the training set that is represented by a row in the matrix X, and a measurement of one of transition t's execution times in that software product that is represented by a corresponding element in vector y. The objective of linear regression is to identify a vector of weights, w, that minimizes the "distance" – as estimated by a *Loss Function* – between Xw and y. In ordinary least squares regression, this corresponds to minimizing $(Xw - Y)^{Transpose}(Xw - Y)$.

such that the derivative of the loss function with respect to the weight vector is equal to zero. We note that using the squared differences as the loss function simplifies the calculations to determine the optimal weight vector, and the sum of squared differences is the most widely used loss function in regression for that reason.

Figure 5.1 illustrates how our approach applies linear regression to estimate the execution time of a transition t in different software products of an Unmanned Air Vehicle SPL. We recall that the Unmanned Air Vehicle SPL, which was introduced in Chapter 2, has three optional features: GPS, Computer Vision, and Additional Motor. A matrix X is used to represent the software products (inputs) in the training set, a vector Y is used to represent transition t's observed execution times (outputs) in the training set, and a vector of weights w is used to represent the regression function f_t.

Each row of matrix X represents the software product (input) of one of t's execution from the training set. The first element of a row is "1" – this "1" is included because the regression function f_t includes a "base weight" w_0 that is added regardless of whether any

input-variable (denoting features in the respective product, in our case) is true or not [1]; The second element of a row is "1" if the product represented by that row has feature GPS and 0 otherwise; the third element of a row is "1" if the product represented by that row has feature Computer Vision and 0 otherwise; and the fourth element of a row is "1" if the product represented by that row has feature Additional Motor and 0 otherwise. The training set is comprised of five input/output pairs, so matrix X and vector Y have five rows. As an example, consider the fourth row of Matrix X, "010" excluding the initial "1", and the fourth row of vector Y, 412. They represent one instance of transition t's execution in product "010" – that is, the product comprising exclusively of feature Computer Vision – taking 412 ms to execute.

As we mentioned earlier, in *ordinarily least squares* regression the objective is to find the optimal vector of weights, $\vec{w} = w_0, \ldots, w_4$, so as to minimize the "squared differences" between transition t's estimated execution times and transtion t's observed execution times. By using the matrix representation, this becomes equivalent to finding the vector of weights that minimizes $(Xw - Y)^{Transpose}(Xw - Y)$.

Accommodating Non-linear Contributions

A linear-regression function is most accurate if the relationship between the input-variable values and the corresponding value of the output variable is linear. However, certain feature combinations may have an outsized positive or negative influence on a transition's execution time because of feature interactions. For example, imagine that transition t takes 20 time units to execute in every software product that includes both feature GPS and feature Computer Vision, whereas it only takes 5 time units to execute in any other software product. Transition t's execution time would depend on the presence of feature GPS and feature Computer Vision video in a non-linear way: t's execution time would be given by the formula $5 + 15 * x_1 * x_2$, where x_1, x_2 are feature variables corresponding to feature GPS and feature Computer Vision.

We incorporate these non-linear relationships into our linear regression models by introducing additional "synthetic" input variables that represent feature combinations. For example, we introduce a "synthetic" input variable $x_{|F|+1} = x_1 \times x_2$ that represents whether both feature GPS and feature Computer Vision are present in a software product. As most

[1] We recall that the regression function f_t is given by $f_t(\vec{x} = [x_1, x_2 \ldots x_{|F|}]) = w_0 + \sum_{i=1}^{|F|} w_i \times x_i$. Consider the product denoted by the fourth row $[1, 0, 1, 0]$ of matrix X in Figure 5.1, which comprises exclusively of feature Computer Vision. The fourth row of the vector obtained by the matrix multiplication Xw is then $1*w_0+0*w_1+1*w_2+0*w_3 = 1*w_0+1*w_2 = w_0+w_2$, which is equivalent to $f_t(\vec{x} = [1, 0, 1, 0])$.

non-functional interactions among features involve only a few features, we introduce a "synthetic" input variables only for each pairwise feature combination. We then apply ordinary least squares with two different sets of input variables: the original input variables (denoted OLS), and the original input variables plus the "synthetic" input variables that represent pairwise feature combinations (denoted OLS-PAIRS).

Overfitting and Cross-Validation

A regression function might learn accidental patterns that occur only in the training set, and these patterns could cause the regression function to be accurate with respect to the training set but not generalize to unseen inputs (e.g., unseen software products). In such case, the regression function is *overfitting* [53] the training set. In the case of linear regression, as we introduce more "synthetic" input variables, the input variables are increasingly interdependent and overfitting becomes more prevalent. In the most extreme case, for each software product in the training set there exists a corresponding input variable that is true in the training set only for that software product; ordinarily least squares could then learn a regression function that is in essence a look-up table – that has perfect accuracy with respect to the training set but that does not generalize to other software products.

Therefore, to decide whether to use OLS or OLS-PAIRS as inputs for our linear regression problem, we must find a balance between the benefits of representing more non-linear relationships versus the risk of overfitting. *K-fold cross-validation* [53] is a machine-learning technique that permits us to determine which of the two inputs provides the best trade-off. In K-fold cross-validation, the training set is partitioned into k equal subgroups. Training is performed on *K-1* groups, while the accuracy of the learning method is assessed on the remaining subgroup, denoted a *validation set*. This process is repeated K times, each time using a different subgroup as the validation set, and the average *validation accuracy* is calculated. As the validation set is not used for training, a method that performs well through overfitting will tend to have a low validation accuracy. The method that obtains the highest validation accuracy is then selected.

5.1.2 Regularized Linear Regression

Regularized linear regression is a variant of linear regression in which simpler regression functions are favored over more complex functions, so that the learnt regression function is easier to understand and does not overfit the training set. To estimate a regression function's complexity, some metric (i.e., a distance function) on its weight vector is used.

The objective is then to identify a weight vector whose corresponding linear regression function minimizes not only the value of the loss function on the training set but also minimizes the value of the weight vector's metric. A parameter, denoted λ, controls the relative importance of each of these two factors and its value is chosen through K-fold cross-validation. More concretely, the objective is to minimize the value of the formula

$$\sum_{(x,y) \in TrainingSet} (f_t(x) - y)^2 + \lambda L(w)$$

where L is a metric (i.e., a distance function) on the weight vector $\vec{w}$. As in standard linear regression, the ideal weight vector is then computed by solving a set of matrix equations.

Our approach uses two types of regularized linear regression: *Least Absolute Shrinkage Operations* (LASSO) regression [103], and *Ridge Regression* [104]. In *LASSO* regression, the metric (i.e., a distance function) on the weights is the sum of the absolute values of the weights (but excluding the fixed constant that is represented by w_0), whereas in *Ridge regression* the metric (i.e., a distance function) on the weights is the sum of the squares of the weights (but excluding the fixed constant that is represented by w_0). Lasso regression tends to zero the weights associated with input variables (e.g., a feature or a pairwise feature combination) that do not have a significant effect on the output variable, whereas Ridge regression tends to reduce the magnitude of those weights but not force them to zero. For each individual learning problem, our approach chooses between linear regression, LASSO regression, and Ridge regression based on their respective validation error during K-fold cross-validation.

To summarize, we set up a learning problem for each transition $t \in T$, and the output is a regression function f_t for each transition t that estimates transition t's execution time in each software product of the SPL.

5.2 Evaluation

We assess the quality of the obtained timed state-transition models by answering the following three research questions:

RQ-1 How accurate are our estimates of a transition's execution time?

RQ-2 How do our estimates of a transition's execution time (in a specific product) compare, in terms of accuracy, with the estimates obtained by Perfume [86], which is a tool that obtains a timed state-transition model by analyzing timed traces of a single software product?

RQ-3 Does the learnt model, projected onto individual software products, accurately estimate the execution times of execution traces? How does the accuracy of such an overall estimate compare to the accuracy of state-of-the-art methods [98], which base their estimates only on which features are present in a software product?

The objective of *RQ-1* is to assess the accuracy of our estimates with respect to ground truth; *RQ-2* compares the accuracy of the cost estimations produced using our approach against the accuracy of the estimates obtained by the closest state-of-the-art method to learn timed behavioural models [86]. *RQ-3* compares our per-transition execution-time estimates against the state-of-the-art in estimating non-functional properties of products in a product line [98]. The state-of-the-art method that we compare our approach against in *RQ-2* is different from the state-of-the-art method used in *RQ-3* because we could not find a method that learns timed state-transition models of SPLs, and we selected the closest most-relevant related work for comparison in each study.

5.2.1 Subject Systems

Our approach requires as input a Software Product Line that includes both a featured transition system model, an executable software system for each software product, and a mapping between transitions in the featured transition system model and their execution in the software system. Most SPLs that are used to evaluate SPL research results do not include a featured transition model [98] or do not include an executable software system [28] Thus, to evaluate our approach we used the following two subject systems: *X264*, an open-source video encoder, and *Autonomoose*, a self-driving car developed at the University of Waterloo. For both subject systems, we had to create the featured state-transition model.

The first subject system is the open-source video encoder *X264*. X264 receives as input a raw video, analyzes each video frame, and encodes each video frame into a compressed representation. X264 accepts multiple runtime options that control how the video will be encoded. For example the option "B-Frame" controls whether a video frame can be compressed by referencing both the preceding and subsequent video frames or only by referencing the preceding video frames. We treat each option of X264 as a software feature and consider X264 an SPL. X264 has twelve features and 2304 software products.

X264 did not have a pre-existing featured state-transition model (or any state-transition model), so we reverse engineered one. To facilitate the understanding of the X264 codebase and ease the reverse-engineering task, we used an early version of X264 (revision 8), which has a much smaller codebase, and we backported features into it from later revisions

(from revisions 134, 251, 318, and 1197). We semi-automatically created a featured state-transition model for X264 using our domain knowledge and aided by the tool Synoptic [8]. We modified the code of X264 to record the executions of transitions from our featured state-transition model and their duration, so that when X264 encodes a video it generates a timed execution trace. The FTS we created for X264 has 21 transitions and 14 states.

The second subject system is *Autonomoose*, which is a self-driving car developed at the University of Waterloo. Autonomoose is a reactive system that periodically executes a sense-analyze-actuate loop. Autonomoose comprises a set of modules, which correspond to software features; the modules communicate with each other through publish/subscribe messaging. Each module executes as an independent process and implements a single unit of functionality. For example, the Localizer module periodically estimates the car's current location and publishes the estimate. The autonomous-vehicle software has been tested both in an actual car and in a simulation environment. Our experiments were performed only in the simulation environment. Autonomoose has six features and 32 software products.

Autonomoose did not have an FTS, so we manually reverse engineered one. As several modules execute in parallel, there is an exponential number of possible interleavings of their executions in each sense-analyze-actuate loop. To facilitate the analysis of Autonomoose, we modified Autonomoose so that modules execute sequentially in each execution loop. We built an FTS model in which the execution of each module corresponds to the execution of a transition. We instrumented Autonomoose to record a timed execution trace each time it executes. The FTS we created for Autonomoose has seven transitions and five states.

5.2.2 Experimental Methods

As explained in Section 5.1, our approach requires as input a set of execution traces from a sample of software configurations. For each subject system, we randomly partitioned all the software configurations into a training set (20% of the software configurations and their corresponding execution traces) which constituted the input to our approach, and a test set (the remaining 80% of configurations and their traces) that we used exclusively to assess the accuracy of the learnt timing models. We chose to use only 20% of the configurations for training because we are interested in determining whether our method can learn accurate timing models from a small subset of the software configurations of an SPL. As the accuracy of the learnt timing function can vary depending on which configurations are part of the training set, we repeated the entire learning procedure ten times using a different training/testing set partition each time. Thus, we learnt ten different timing models for each transition, separately evaluated the accuracy of each one of these timing models, and

reported the mean values of the accuracy indicators when evaluating our learnt models. The standard deviation of the accuracy indicator across the ten different training sets was small: in average it was 0.82% for X264 and 35.1% for Autonomooose.

To generate the execution traces, we executed every configuration of X264 on three videos taken from the MPEG-4 test set, and executed each configuration/video pair ten times. We executed the X264 configurations on a computer with 32 gigabytes of of memory, a 2.8GHz AMD Opteron Processor 2439 SE with 6 cores, running Ubuntu version 16.04.5. Similarly, we executed every configuration of Autonomoose in three driving scenarios (e.g., test cases) and executed each configuration/scenario pair ten times. We executed the Autonomooose configurations on a computer with 16 gigabytes of memory, a 3.3GHz AMD Processor FX-8370E with 8 cores as its CPU, running Ubuntu version 16.04.5. We should note that we executed every possible configuration of the subject systems to obtain a thorough evaluation of our learnt model's accuracy (that is, the test set comprised every configuration that was not part of the training set); whereas when applying our method in practice, only the configurations that comprise the training set would be executed.

We chose to randomly select the software configurations that comprise the training set. Alternatively, we could have followed the approach of Siegmund et. al. [98] in selecting the software configurations such that they include configurations with and without each feature, and with and without each possible pairwise feature-combination, to facilitate learning a more accurate model. However, that approach assumes that one is able to choose the software products that comprise the training set and this might not always be possible; in some situations we are given a set of existing execution traces for a set of software products. We think that using a random set of configurations as the training set better reflects the challenges of learning in such a situation. Moreover, Kaltenecker et al. [65] showed that selecting a random set of configurations as the training set results in learning a more accurate model than using heuristics, such as done by Siegmund et al. [98].

5.2.3 RQ-1 Accuracy of Transition Time Estimates

The goal of this research question is to evaluate the accuracy of the learnt per-transition execution-time estimates with respect to ground truth. We assessed the accuracy of the execution-time estimates, for each transition t, by comparing the estimated times against each transition's actual execution times, for all instances of the transition in all execution traces of all configurations in the test set. We chose to use the *mean absolute error* (MAE) as our accuracy indicator because it is an easy-to-interpret measure of accuracy. The mean

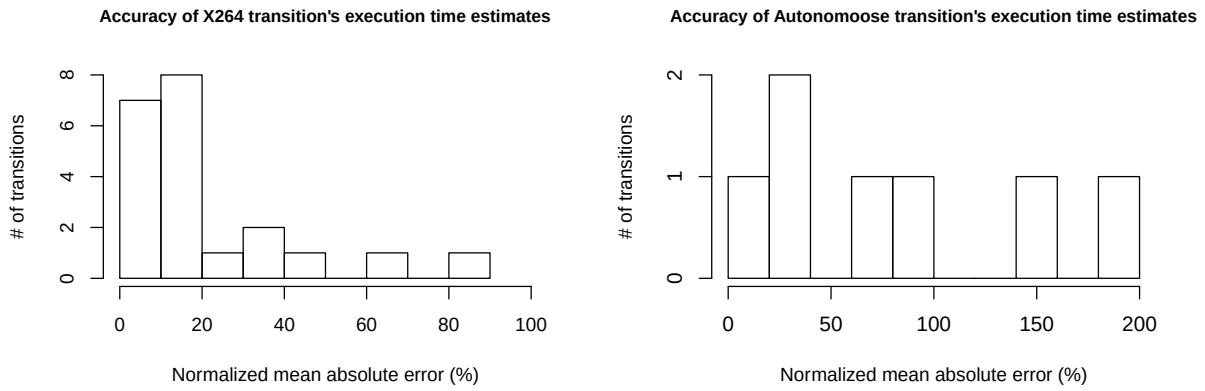


Figure 5.2: Histograms showing the average accuracy of our per-transition execution-time estimates for Autonomoose and X264.

absolute error represents how far off, on average, transition t's estimated execution time is from its actual execution time. To obtain a measure of the relative error, we calculate *normalized mean absolute error* (NMAE) of our execution time estimate for a transition t by dividing the transition's mean absolute error by the average execution-time of transition t in the entire test set [2].

Our method is capable of learning timing functions with a NMAE of less than 30% for 16 out of the 21 transitions of X264 but for only 2 out of the 7 transitions of the Autonomoose software system. Figure 5.2 shows a histogram that summarizes the accuracy of the learnt timing functions across all transitions. Each bin represents the number of transitions whose corresponding timing function had a normalized mean absolute error in a certain range (e.g., 0% to 10%, 10% to 20%, and so on).

Table 5.1 lists the NMAE for the transitions in which our method obtained the best, median and worst accuracy (taken over all executions of each transition), listing the mean and the standard deviation for each NMAE. Thus, the average NMAE of our method's execution time estimates for X264's transitions was 21.9% whereas it was 80.3% for Autonomoose's transitions. The accuracy of our method's timing predictions varies substan-

[2] An alternative approach would have been to use the average *prediction error* ([50] and [98]), in which the estimation error is divided by the actual value of the quantity being estimated (i.e., the time taken by an execution of a transition in our case). We did not use the average prediction error because it is not well-defined when the estimated quantity has a value of zero as was the case for the execution of some transitions in both Autonomoose and X264.

Transition / Software System	X264	X264	Autonomoose	Autonomoose
	Training[1]	Testing[1]	Training[1]	Testing[1]
Transition with best NMAE	3.8% ± 9.8 %	3.8 % ± 6.0 %	16.0 % ± 36.2 %	18.0 % ± 21.6%
Transition with median NMAE	17.3 % ± 17.9 %	17.2% ± 17.7 %	23.2 % ± 21.5%	63.3 % ± 72.5 %
Transition with worst NMAE	81.7 % ± 162.1%	81.7 % ± 170.0 %	192.0 % ± 730.9 %	193.0 % ± 732.7 %
Average NMAE across all transitions	21.8 % ± 20.1 %	21.9 % ± 20.1 %	72.2 % ± 72.4 %	80.3 % ± 66.8 %

[1]normalized mean absolute error ± standard deviation

Table 5.1: Accuracy (normalized mean absolute error) of per-transition execution-time estimates for Autonomoose and X264.

tially based on the transition whose execution time is being predicted. Specifically, the standard deviation of the NMAE, measured across all transitions, was 20.1% for X264 and 66.8% for Autonomoose. We think that our approach produces more accurate estimates for the X264 system than for the Autonomoose system because the number of configurations of Autonomoose is too small for our approach to effectively learn how each feature impacts a transition's execution times (Autonomoose has a total of 32 configurations whereas X264 has 2034 configurations). In Section 5.2.5 we discuss in more detail possible reasons why our approach is more accurate on the X264 system than on the Autonomoose system.

5.2.4 RQ-2 Comparison against Accuracy of Perfume Transition-Time Estimates

The goal of this research question is to compare the accuracy of the models obtained by our approach against the accuracy of models obtained by state-of-art methods. There is no other tool that learns timed behaviour models of software product lines. The closest related work is Perfume [86]: a tool that learns a timed behaviour model – a timed state-transition model – from a set of timed execution traces of a software system (i.e., a single product). Thus, we compare the accuracy of the models learned by our method against the accuracy of the models learned by Perfume.

Perfume and our method differ in multiple ways that complicate their comparison:

- **Expect slightly different input.** Perfume is not product-line aware but instead analyzes execution traces from a single software product and outputs a single corresponding timed state-transition system. In contrast, our method takes as input an untimed state-transition model, analyzes the execution traces of a *collection* of software products, and outputs a single timed model that estimates the timing of transitions in all the products of a product line.

- **Produce slightly different outputs.** Perfume infers not only the execution time of each transition but also infers the state-transition model itself. Specifically, Perfume starts by creating a timed state-transition model whose transitions are exactly the set of steps in the input traces (i.e., a tree-like timed state-transition model), and then iteratively merges states and transitions based on which other transitions appear after them and how long they take to execute. The result is typically a model that is less compact than a model created by hand. As a result, a single transition in our timed featured transition system can correspond to multiple transitions in the timed state-transition output by Perfume. Moreover, Perfume outputs either a number or

an interval as the estimated execution time for each transition and a relative count that represents how often the transition was executed in the input trace, whereas our method outputs a single number as the estimate for the execution time of a transition in a given product.

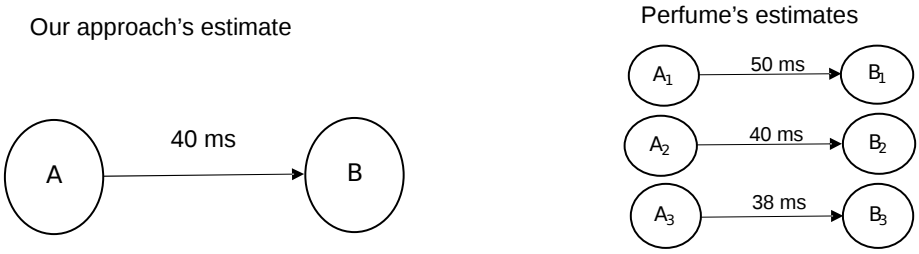
In order to compare the two approaches despite their mismatches in inputs and outputs, the evaluation design needed to accommodate these mismatches.

Firstly, Perfume learns models of a single product, whereas our method learns a product-line model. Therefore, we randomly selected ten software configurations of each subject system from the test set, with the intent of comparing for each configuration the Perfume learnt model against the corresponding projection of our method's learnt model projected onto that configuration's product.

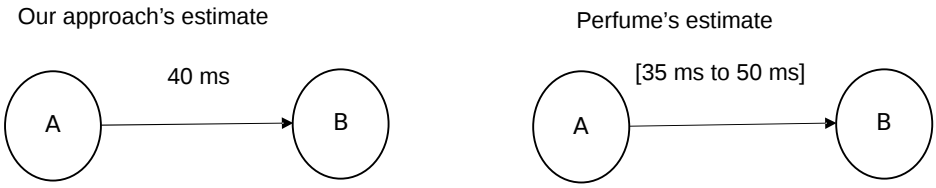
Secondly, we executed Perfume on the sets of timed execution traces associated with each configuration, so that Perfume learnt a timed behaviour model for each selected configuration. We attempted to encourage Perfume to produce more compact models. Because Perfume learns the structure of a state-transition model as well as transitions' execution times, it requires as input a large number of execution traces. Rather than feed Perfume a number of long traces, we split execution traces into a large set of sub-traces that represent executions of a key section of the subject system based on our domain knowledge of the subject system. For example, in the X264 case study, the sub-traces correspond to the analysis of a video frame to decide how to encode it, whereas in the Autonomoose case study the sub-traces correspond to a full sense-plan-act loop. Given an input of shorter execution subtraces, Perfume is better able to generate models that identify multiple executions of the same code as being instances of the same transition, so that it is easier to identify which transitions in the Perfume models correspond to transitions in our approach's models.

Thirdly, to compare, for each of the ten selected configurations, the projection from the learnt timed FTS against Perfume's learnt timed-transition model, we first had to identify for each transition in our timed FTS that is present in one of the ten configurations, the set of corresponding transitions in the timed-transition model output by Perfume for that configuration. A transition t in our approach's model could correspond to either a single transition in Perfume's model or to multiple transitions in Perfume's model. Moreover, each transition in a Perfume model could be labelled with either a single number or labelled with an interval. To enable a comparison, for each transition t in our approach's model we computed a single weight (the mean and standard deviation) for the weights on the corresponding transition(s) in the Perfume model. Specifically, the four cases are as follows:

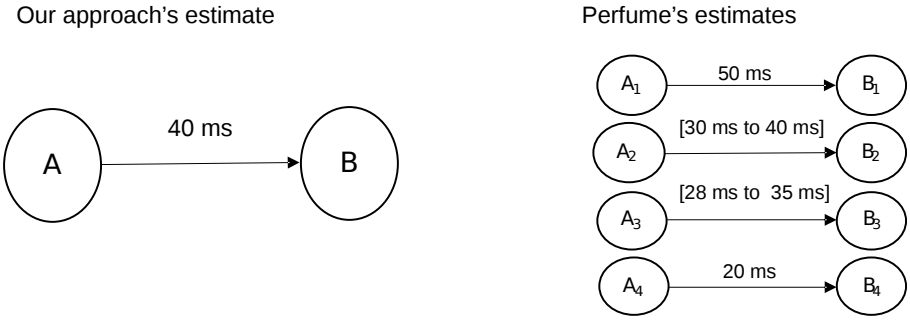
(a) The case where a transition in the FTS corresponds to a single transition in Pefume's model that has a single numeric weight.



(b) The case where a transition in the FTS corresponds to multiple transitions in Pefume's model, each with a different numeric weight.



(c) The case where a transition in the FTS corresponds to a single transition in Pefume's model that has an interval weight.



(d) The case where a transition in the FTS corresponds to multiple transitions in Pefume's model, with some transitions having a numeric weight and some having an interval weight.

Figure 5.3: Ilustration of the relationships between a transition's execution-time estimate from our approach and from Perfume.

- A transition t in our approach's model may correspond to a single transition in Perfume's model that is labelled with a single numeric weight. Figure 5.3a illustrates this case by showing a transition in Perfume's model with a numeric weight of 50 milliseconds that corresponds to transition t in our approach's model. In this case, the mean of Perfume's execution-time estimates for transition t is equal to its numeric weight and its standard deviation is equal to zero.

- A transition t in our approach's model may correspond to multiple transitions in Perfume's model, each with a different numeric weight. Figure 5.3b illustrates this case by showing three transitions in Perfume's model, each with a different numeric weight, that correspond to transition t in our approach's model. To enable a comparison, we compute the mean and standard deviation of the transition estimates weighted by the number of times the transition executes in the traces. For example, if each of the Perfume model's transitions in Figure 5.3b executed twice in the traces, we would compute the mean and standard deviation of the bag $\{50, 50, 40, 40, 38, 38\}$ to obtain Perfume's estimate for that transition.

- A transition t in our approach's model may correspond to a single transition in Perfume's model that is labelled with an interval weight. Figure 5.3c illustrates this case by showing a transition in Perfume's model with an interval weight of between 35 and 50 that corresponds to transition t in our approach's model. Intervals are more difficult to compare against. We make the simplifying assumption that the interval represents a uniform distribution of possible weights, and we compute a mean and standard deviation based on n uniformly distributed numbers in the interval where n is the number of times the transition executes in the input traces. For example, if the transition in 5.3c, which is labelled with an interval from 35 to 50 milliseconds, executed four times in the input traces, we would compute the mean and standard deviation of the set $\{35, 40, 45, 50\}$.

- A transition t in our approach's model may correspond to multiple transitions in Perfume's model, some with a numeric weight and some with an interval weight. Figure 5.3d illustrates this case by showing four transitions in Perfume's model that correspond to transition t in our approach's model, with two of those transitions having a numeric weight and the other two transitions having an interval weight. We generate a single mean weight and standard deviation as follows. For each of the corresponding transitions (in the Perfume model) that is labelled with a single numeric weight, we generate a collection of n instance of that numeric weight, where n is the number of times that transition executes in the input traces. For each corresponding transition (in the Perfume) that is labelled with an interval weight, we

generate a collection of n numbers that are uniformly distributed within the interval, where n is the number of times the transition executes in the input traces. We then calculate the mean and standard deviation of the numbers within these collections to obtain the mean and standard deviation of Perfume's execution-time estimates for transition t. For example, consider the transitions (in the Perfume model) in Figure 5.3d, and assume that the transition that is labelled with numeric weight "50" is executed three times, that the transition that is labelled with numeric weight "20" is executed two times, that the transition that is labelled with the interval "30ms to 40ms" is executed two times, and that the transition labelled with the interval "28ms to 35ms" is executed three times. Then we would compute the mean and standard deviation of the bag of weights $\{50, 50, 50, 20, 20, 32.5, 47.5, 28, 31.5, 35\}$ to obtain Perfume's execution-time estimate for transition t.

In this manner, for each transition and software configuration pair, we obtain the mean and standard deviation of Perfume's execution-time estimates for it. We can then compare it against our method's estimates for the same transition and software configuration pair.

In Figure 5.4 we show our method's and Perfume's execution-time estimates for the transitions from three of the ten software configurations: the software configurations in which our method estimates had the best, median, and worst accuracy. For each transition, Perfume's mean execution-time estimate is marked with a red dot and the red error bar represents the standard deviation; and the grey error bar represents Perfume's maximum and minimum execution-time estimates for that transition. Our method's execution-time estimate for each transition is represented by a black X.

We can observe that in the case of X264 most of our execution-time estimates are within one standard deviation of Perfume's estimates. In contrast, in the case of Autonomoose, we observe that many of our estimates are more than one standard deviation away from Perfume's estimates. However, the best and median case of Autonomoose look very similar, so its worst case might just be an outlier.

5.2.5 RQ-3 Accuracy of trace time estimates

As we discussed in Subsection 5.2.3, the per-transition execution time estimates provided by our approach are not very accurate or precise. Other approaches [50, 65, 82, 98, 99] learn a static estimate of how long will a software product take to execute over a given execution trace (input) based on which features are present in the software product instead of computing an estimate based on which parts (transitions) of the system the trace exercise

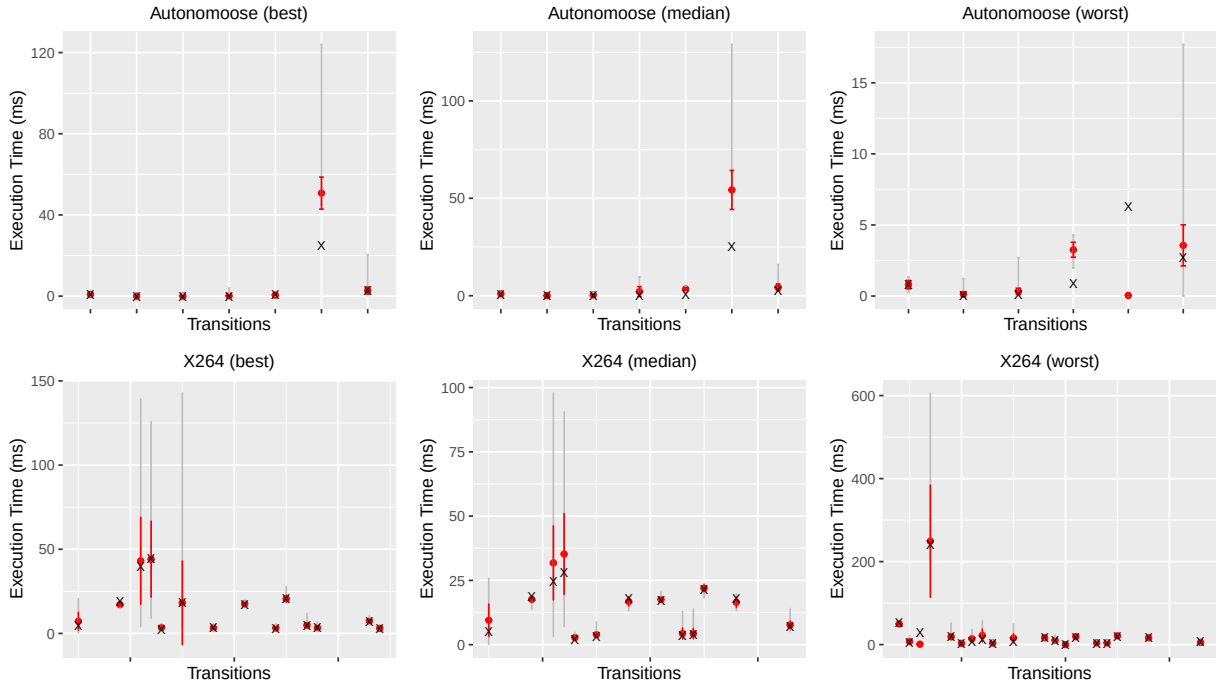


Figure 5.4: Graph showing both Perfume and our approach's execution-time estimates for the transitions in three software products (the software products for which our approach had best, median, and worst accuracy). Perfume's mean execution-time estimates are marked as a red dot and the standard deviation is represented by a red error bar. Our approach's execution-time estimates are marked with a black X. The grey error bars represent Perfume's minimum and maximum execution-time estimates.

Table 5.2: Normalized mean absolute error of the overall execution-time estimates for X264 and Autonomoose.

Method / Software System	X264	Autonomoose
Based on per-transition estimates	0.62%±0.47%	42.15%±45.65%
Based on Performance-Influence model [98]	7.49%± 8.38%	29.37%±37.49%

and the execution-time estimates of those parts of the system. We hypothesize that we can leverage our per-transition execution time estimates to obtain a more accurate estimate of the overall execution time of a software product over an execution trace (e.g, encoding a video). If this hypothesis would be true, it would suggest that it could be beneficial to use finer-grained weighed model instead of Performance-Influence model to estimate the execution times of different software products. We test this hypothesis by comparing the accuracy of estimates of the execution time of a software product over an execution trace obtained by the Performance-Influence model [98], against estimates derived from our per-transition execution-time estimates.

As mentioned earlier, the Performance-Influence model [98] is a representative approach of the methods that estimate the overall execution time of a software product over an execution trace based on which features are present in the software product. Because our method provides per-transition execution-time estimates, we need to compute the estimate of a software product over an execution trace: by multiplying the number of executions of each transition in the trace by the estimated execution time of that transition. We then compare the accuracy of such estimates against the accuracy of the estimates obtained by Performance-Influence model [98] and against ground truth (the actual execution times).

We used the same two subject systems, Autonomoose and X264, as in the other research questions in this comparison. We recorded the overall execution time of X264 on each of three videos (input traces) for each product in the training set and testing set, and similarly we recorded the overall execution time of Autonomoose on each of three driving scenarios (input traces) for each product in the training set and testing set. We executed the learning algorithm of the Performance-Influence model approach [98] on the training set to obtain regression functions that estimate the execution time of each software product on each one of the respective three input traces. Similarly, our approach's per-transition execution time estimates were learnt using only the training set. We assessed the accuracy of our approach's overall execution-time estimates compared against ground truth and compared against the Performance-Influence model [98] estimates with respect to the execution times over the input traces of the software products that comprise the testing set. As in *RQ-1*, we use the *normalized mean absolute error* (NMAE) as our accuracy indicator because it

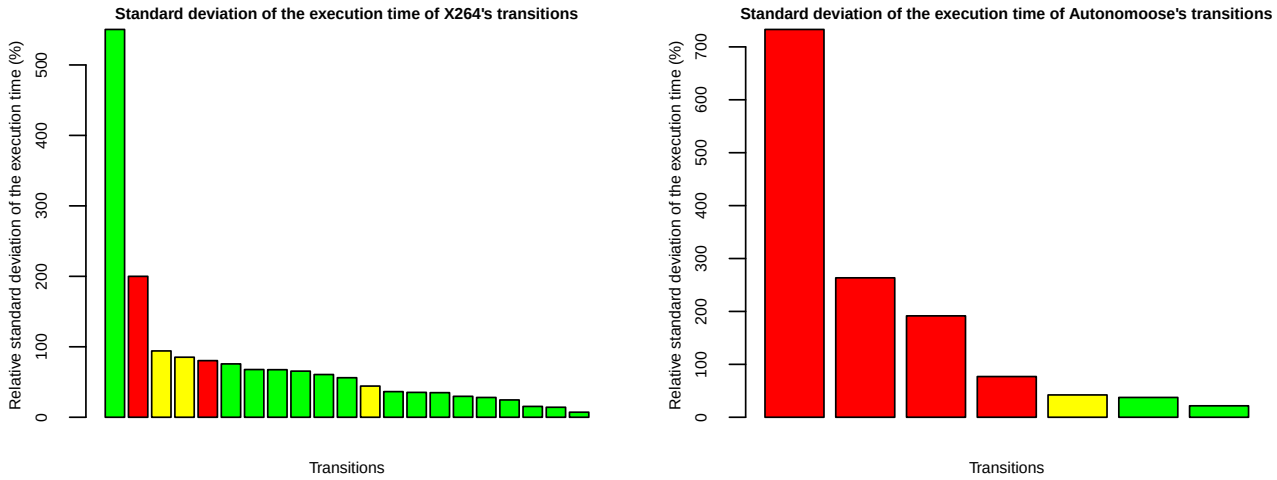


Figure 5.5: Bar chart of the standard deviations of the execution times of transitions in the subject systems: X264 and Autonomoose. Along the x-axis, transitions are listed from left to right in decreasing order of the standard deviations of their execution times. The bar color represents the accuracy of our execution-time estimates for that transition: green represents a NMAE of less than 30%, yellow represents a NMAE between 30% and 50%, and red represents a NMAE greater than 50%.

is an easy-to-interpret measure of accuracy. We calculate the mean NMAE across all the executions of all software products in the testing set and across all inputs (i.e., all three videos for X264 and all three driving scenarios for Autonomoose).

A summary of the results of this comparison is shown in Table 5.2 . Specifically, we show the normalized mean absolute error for both our method and the Performance-Influence model approach. We show the average NMAE of the estimates of the execution time of software products that comprise the testing set. Our method can accurately estimate how long a X264 software product will execute (e.g., encode a software video), but our method estimates are not very accurate for Autonomoose. Our method obtains a normalized mean average error of only 0.62 % for X264, whereas our method obtains a normalized mean average error of 42.15% for Autonomoose.

Obtaining much better accuracy for X264 than for Autonomoose is expected because we have more accurate per-transition timing estimates of X264's transitions than of Autonomoose's transitions.

The accuracy of our per-transition estimates varied substantially across different transitions. This was the case for both subject systems. We hypothesize that such variation is

caused by the variation in the standard deviation of the execution times of each transition. Figure 5.5 shows the relative standard deviations of the execution times of each transition; the transitions are arranged along the x-axis in decreasing order of the standard deviation in the values of their execution times. Each bar is color coded according to the accuracy in our estimate of the transition's weight: green indicates a NMAE of less than 30%, yellow indicates a NMAE between 30% and 50%, and red indicates a NMAE greater than 50%. For both X264 and Autonomooose we observe that as the relative standard deviation of the execution times of transitions increases, the error in our execution-time estimates (e.g. the NMAE) increases. This correlation seems intuitive – if there is a lot of variation in the cost of a transition's execution due to context, the input data being processed, etc. then it is difficult to learn accurate values for the transition's weight. Whether there is high variance in the cost of a transition's execution varies by SPL, product, and transition.

We hypothesize that we were able to obtain much higher accuracies in our estimates of the weights on transitions in the X264 model than in our estimates of the weights on Autonomooose's transitions because the number of software configurations in the Autonomoose SPL is too small for our method to effectively learn how different features impact the execution time of each transition (the Autonomoose SPL has 32 products whereas the X264 SPL has 2034 products).

It is noteworthy that although our estimates of each transition's weight have relatively low accuracy, our estimates of the overall execution times of complete traces (e.g., encoding a video for the X264 case study) are relatively good. This could be due to the errors in the estimates cancelling each other out when combining the estimates for all executed transitions into an estimate for a single execution trace.

5.2.6 Threats to validity

Our results are based on only two software product lines, and the conclusions might not generalize to other software product lines. To mitigate this threat, we chose two SPLs from two different domains. Both SPLs had fewer than thirteen features, so our results might not generalize to much larger SPLs. We had to reverse engineer the FTSs for both SPLs, and this might affect whether our results are valid for an FTS that is created as part of the development of an SPL.

The results that we obtained could have been affected by chance (e.g., a “lucky” training set or an abnormal execution of the subject system). To minimize this threat, we repeated all experiments ten times with a different training set and testing set. For the same reason, we executed each SPL ten times on each of the three different “input traces” when

generating the timed execution traces. However, we did not vary the relative size of the training set compared to the evaluation set, and this could have affected the results. Also the execution traces used to generate the test and training set might not cover all possible executions paths of the subject systems.

The configurations that comprised the training sets were selected randomly. A special or systematic sampling of the configuration space might have provided more accurate learning – for example, selecting the configurations with the minimum/maximum number of features to be part of the training set. Thus, our conclusions might not generalize to techniques that perform "smart sampling" of the SPLs configurations to obtain the training set.

5.3 Related Work

There exist many methods that automatically extract untimed behaviour models from a system's execution traces [7, 8, 45, 75, 76, 77]. For example, Synoptic [8] infers a state-transition model from a set of observed execution traces, while also ensuring that the state-transition model satisfies a set of temporal invariants that are mined from the observed execution traces. Synoptic starts with a given state-transition model whose set of transitions is the set of steps in the observed traces, and then iteratively merges states and transitions to obtain a more compact and general state-transition model. More recently, Perfume [86] extends Synoptic to infer a timed state-transition model from a set of timed execution traces. Perfume works by mining a set of timed properties (e.g., by instantiating timed requirement patterns [71]) that are valid across all execution traces. Perfume then searches for a compact timed transition system that can generate traces similar to the observed traces and that satisfies the mined timed properties. Similarly, Timed K-Tail [92] extends GK-Tail [76] to generate a timed automaton from a set of timed execution traces. None of these methods have been adapted to analyze execution traces from a set of software products [98].

The work of Siegmund et al. [98] learns a Performance-Influence model for an SPL that predicts how long each software product will execute by assigning (through learning) a static weight to each feature and to some feature combinations that represents the feature's contribution to the product's execution time. The estimated execution time for a software product becomes the sum of the weights of all features and feature combinations that are present in that software product. We note that their method learns the execution of a software product over a "representative task" – such as encoding a video, or compressing a file. Their approach uses a coarser level of granularity than our method,

as it learns the impact that each feature or feature combination will have on the overall execution time of a software product, whereas our method learns the impact that a feature or feature combination will have on the execution time of each transition. Their approach ignores the effect that changing user-input or changing environmental conditions can have the execution time of an SPL at runtime. Subsequent papers [83, 84, 85] focus on finding optimal software configurations based on these coarse-grained models of a product's execution time.

5.4 Conclusions

Analysts need accurate and precise timed models of systems and SPLs to assess whether the modelled systems satisfy timing properties. Learning timed behaviour models from execution traces for an SPL is a very challenging task, and the results of our approach is mixed with respect to the accuracy of our learnt models. We showed that our method seems promising for SPLs with a large number of products, and whose transitions' execution times do not have excessive variance across repeated executions, and whose transitions' execution times are a linear function of the features present in a software configuration. Through our analysis of RQ-3, we showed that these learnt timed behaviour models can potentially facilitate analysis of the timing behaviour of SPLs, by helping to estimate how long will an SPL configuration take to execute a certain execution trace.

Chapter 6

Conclusions and Future Work

Quantitative analyses of weighted transition systems permit estimating the quality-attribute values that a software system will exhibit during its execution. Quantitative analyses can be computationally intensive and the number of products in an SPL grows exponentially with the number of features in the SPL; thus analyzing every product in an SPL is challenging. In this thesis, we investigated the efficiency of performing family-based quantitative analyses of quality-attribute values that each product of an SPL exhibits.

First, we considered quality-attribute values for a dynamic software product line that is executing in a rapidly changing environment. As the environment changes, the product that is optimal with respect to its quality-attribute values might also change. Thus, we proposed and implemented a family-based trace-checking analysis that estimates, at runtime, the quality-attribute values that the different products of a dynamic software product line would have exhibited over recent inputs, to facilitate reconfiguration of the dynamic software product line to the optimal product. We obtained mixed results with respect to the performance benefit of such family-based analysis compared to individually analyzing each software configuration. Our results indicate that the family-based trace-checking analysis, without any data abstraction, is not faster than individually analyzing each product for two of the three case studies in which many of the configurations exhibited similar but slightly different quality-of-service values. Because many configurations would yield similar but slightly different quality-of-service values, there was little sharing of partial-analysis results in those two case studies. However, we showed that by adding a simple data abstraction, which categorized transitions into high-cost and low-cost transitions, the performance of the family-based trace-checking analysis improved such that it was between 1.4 and 7.7 times faster than individually analyzing each product. This suggests that abstraction over the values of the quality attributes is key to permit our

family-based trace-checking analysis to reuse partial-analysis results, and thus provide a speedup compared to analyzing each product individually.

Second, sometimes what is important for the users of an SPL is the worst-case long-term average value of a quality attribute (e.g., because it represents the long-term rate of energy consumption of the system). Specifically, the maximum limit-average cost of executing a weighted state-transition model can represent the worst-case long-term average value of a quality attribute over an infinite execution of the system. Because computing the maximum limit-average cost of a product is computationally intensive, we developed a family-based analysis that simultaneously computes the maximum limit-average cost for each software product in an SPL. We assess its performance against analyzing each product individually in two case studies. Again, we obtained mixed results: our family-based approach increased the speed of analysis by a factor of two in one SPL with a large number of products, a small number of control states, and whose products shared the same set of strongly connected components; whereas our analysis was slower than the product-based approach in the other SPL that had fewer products but included a rich control behaviour. Our results suggest that our family-based analysis will perform better in SPLs in which many products share the same set of strongly connected components. It might be beneficial to use partial results from our family-based analysis (e.g., how much sharing of state's "finishing times" there is across software products) to decide whether to proceed with a family-based analysis or a product-based analysis for a specific SPL.

Third, analyses of a system's quality attributes, including our analyses, require as input a weighted behaviour model of an SPL. But manually developing an SPL's weighted behaviour model is tedious and error-prone: it requires creating a model that estimates the weight associated with a transition's execution for each product of the SPL. We explored how to learn the weights for an SPL's weighted behaviour model (i.e., a weighted featured transition system model) from a set of execution traces from a sample of the software products of an SPL. Specifically, we applied supervised learning techniques to learn, for each transition, a regression function that estimates how much time the transition will take to execute in each software product that includes the transition. We assessed the accuracy of the learnt transition execution-time estimates on two SPLs, over which the accuracy of our learnt model ranged from a mean error of 3.8% to a mean error of 193.0%. Our method performed best for those transitions whose execution-times (1) have low variance across repeated executions of the transition in the same product and (2) is a linear function of the features present in a software product. We also compared the accuracy our learnt per-transition execution time estimates against the estimates from a state-of-the-art tool (Perfume [86]) that extracts a weighted transition system model for an individual software product by analyzing its individual execution traces. We showed that we could achieve

an accuracy comparable to the accuracy obtained by Perfume, even though we base our estimates on the execution of a *sample* of software products and not on the execution of the software product from which Perfume was extracting the weighted transition model. We also compared the accuracy of of our learnt weighted featured transition system models' estimates against the state-of-the-art Performance-Influence model [98] by comparing their estimates of the time to execute an execution trace, and against ground truth. The accuracy of our learnt weighted featured transition system models' estimates of the time to execute an execution trace was similar to the accuracy of the estimates obtained by the state-of-the-art Performance-Influence model [98].

In summary, the main contributions of this thesis are:

- A family-based trace-checking algorithm (and its implementation) that analyzes the quality of service that each configuration of a DSPL would have exhibited over recent system inputs.
- An evaluation of the efficiency of such an analysis on three DSPL case studies taken from the literature. Our results suggest that abstraction over the values of the quality-attribute values is key to enable efficient family-based trace checking analysis of the quality attributes.
- Improvement over our initial trace-checking algorithm by applying a simple data abstraction over the values of quality attributes.
- A family-based algorithm (and its implementation) that analyzes a model of an SPL and computes the worst-case limit-average value for a quality attribute.
- An evaluation of the speed-up of such a family-based algorithm versus the product-based algorithm. Our results suggest that our family-based analysis will perform best for SPLs in which many of the products share the same set of strongly connected components.
- We present and evaluate for the first time a method to learn timed behaviour models for software product lines.
- An evaluation of the quality of the learnt weighted featured transition systems models of an SPL. Our results indicate that the accuracy of the learnt models is best for SPLs in which there is a linear relationship between its transitions' execution times and the features that are present in a software configuration, and for SPLs in which its transitions' execution times do not have excessive variance across repeated executions.

- A comparison of our method to the state-of-the-art methods for extracting Performance-Influence models of a software product line and methods for learning timed behaviour models from timed execution traces.

The family-based analysis of a quality attribute of an SPL can in some cases be faster than the corresponding analysis of each individual software product. However, a key challenge is the difficulty in reusing partial-analysis results across different software products because the products can have slightly different quality-attribute values. Data abstraction of quality-attribute values might improve the performance of a family-based analysis of an SPL's quality attributes by making many products share the same abstract quality-attribute values, and thus facilitate the reuse of partial-analysis results. A promising line of future inquiry is to explore the role that data abstraction over quality-attribute values can play in improving the performance of family-based analyses of an SPL's quality attribute. Specific types of quality attributes will evolve in certain unique ways (e.g., time increases monotonically, the battery level of a system has an upper bound). We think that, by studying specialized data abstractions for different types of quality attributes, researchers will be able to ease the task of identifying suitable abstractions for quality attributes that permit faster but also accurate analyses. It would be beneficial to study the trade-off between abstraction over quality-attribute values against the accuracy of the analysis results with respect to identifying the configuration with optimal quality-of-service.

A second line of future work is to further assess whether our family-based analysis of the maximum limit-average cost performs best in SPLs that have many products that share the same set of strongly connected components as more SPL models becomes available. We could also assess whether using partial-results from our family-based analysis to determine on which set of products to perform a family-based analysis (e.g., only on those products that share most of their SCCs) and individually analyze the rest of the products would speed up the analysis.

A third line of future work that extends our work on learning timed behaviour models of software product lines would be to identify the causes for variance among the execution times of the same transition even in the same product. This variance could be caused by: i) noise in the measurement of the execution time, ii) varying complexities of the transition's action based on inputs, iii) varying execution times due to prior actions or transitions, environmental conditions, or input trace. This future exploration could help to identify more precisely in which type of SPLs our learning approach will perform best.

Additionally, it might be possible to incorporate domain knowledge about which non-linear variables (e.g., the presence of specific feature combinations) are likely to impact a transition's execution time, and incorporate such knowledge into the learning algorithm.

Manually selecting which feature combinations to incorporate as input into the learning algorithm might result in better accuracy compared to including every possible pairwise feature combination as input to the learning algorithm.

References

[1] *Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, 2010.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor.Comput.Sci.*, 126(2):183–235, apr 1994.

[3] J. Andersson and J. Bosch. Development and use of dynamic product-line architectures. *IEE Proceedings - Software*, 152(1):15–28, Feb 2005.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[5] Gerd Behrmann, Alexandre David, Kim G. Larsen, John Hakansson, Paul Petterson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems*, QEST '06, pages 125–126, Washington, DC, USA, 2006. IEEE Computer Society.

[6] Shoham Ben-David, Baruch Sterin, Joanne M. Atlee, and Sandy Beidu. Symbolic model checking of product-line requirements using sat-based methods. In *ICSE*, pages 189–199. IEEE Press, 2015.

[7] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Transactions on Software Engineering*, 41(4):408–428, April 2015.

[8] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 267–277, New York, NY, USA, 2011. ACM.

[9] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag, Berlin, Heidelberg, 2006.

[10] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[11] Udi Boker, Krishnendu Chatterjee, Thomas A. Henzinger, and Orna Kupferman. Temporal specifications with accumulative values. *ACM Trans. Comput. Log.*, 15(4):27:1–25, 2014.

[12] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. 24(3), 1992.

[13] Radu Calinescu, Lars Grunske, Marta Kwiatkowska, Raffaela Mirandola, and Giordano Tamburrelli. Dynamic qos management and optimization in service-based systems. *IEEE Trans. Softw. Eng.*, 37(3):387–409, 2011.

[14] Javier Camara, Antonia Lopes, David Garlan, and Bradley R. Schmerl. Impact models for architecture-based self-adaptive systems. In *Formal Aspects of Component Software - 11th International Symposium, FACS 2014*, 2014.

[15] Sérgio Vale Aguiar Campos, Edmund M. Clarke, Wilfredo R. Marrero, and Marius Minea. Verifying the performance of the pci local bus using symbolic techniques. In *Proceedings of the 1995 International Conference on Computer Design: VLSI in Computers and Processors*, ICCD '95, pages 72–78, Washington, DC, USA, 1995. IEEE Computer Society.

[16] Pavol Černý, Martin Chmelík, Thomas A. Henzinger, and Arjun Radhakrishna. Interface simulation distances. *Theor. Comput. Sci.*, 560:348–363, 2014.

[17] Pavol Černý, Thomas A. Henzinger, and Arjun Radhakrishna. Simulation distances. *Theor. Comput. Sci.*, 413(1):21–35, January 2012.

[18] Pavol Černý, Thomas A. Henzinger, and Arjun Radhakrishna. Quantitative abstraction refinement. In Giacobazzi and Cousot [48], pages 115–128.

[19] Krishnendu Chatterjee, Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Polynomial-time algorithms for energy games with special weight structures. *Algorithmica*, 70(3):457–492, November 2014.

[20] Krishnendu Chatterjee, Andreas Pavlogiannis, and Yaron Velner. Quantitative interprocedural analysis. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 539–551, New York, NY, USA, 2015. ACM.

[21] Shang-Wen Cheng and David Garlan. Stitch: A language for architecture-based self-adaptation. *J. Syst. Softw.*, 85(12):2860–2875, December 2012.

[22] Philipp Chrszon, Clemens Dubslaff, Sascha Kluppelholz, and Christel Baier. Family-based modeling and analysis for probabilistic systems — featuring profeat. In *Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering - Volume 9633*, pages 287–304, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[23] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[24] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Sci. Comput. Program.*, 80:416–439, 2014.

[25] Andreas Classen, Maxime Cordy, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Model Checking for Software Product Lines with SNIP. *International Journal on Software Tools for Technology Transfer*, 2012.

[26] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-Francois 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, August 2013.

[27] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *Proc. of the 33rd International Conference on Software Engineering*, ICSE '11, 2011.

[28] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *ICSE*, pages 335–344. ACM, 2010.

[29] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA, 2001.

[30] Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Provelines: A product line of verifiers for software product lines. In *Proc. of the 17th International Software Product Line Conference, SPLC*, 2013.

[31] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Behavioural modelling and verification of real-time software product lines. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 66–75, 2012.

[32] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 472–481, 2013.

[33] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, 2nd edition, 2001.

[34] Pedro R. D'Argenio and Hernán C. Melgratti, editors. *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, volume 8052 of *LNCS*. Springer, 2013.

[35] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. TACAS'08/ETAPS'08, page 337340, Berlin, Heidelberg, 2008. Springer-Verlag.

[36] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 411–420, May 1999.

[37] Niklas En and Niklas Srensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*. Springer, 2003.

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

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

[40] Miguel Felder and Angelo Morzenti. Validating real-time systems by history-checking trio specifications. In *Proc. of the 14th International Conference on Software Engineering*, ICSE '92, 1992.

[41] A. Filieri, G. Tamburrelli, and C. Ghezzi. Supporting self-adaptation via quantitative verification and sensitivity analysis at run time. *IEEE Transactions on Software Engineering*, 42(1):75–99, Jan 2016.

[42] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. Run-time efficient probabilistic model checking. In *Proc. of the 33rd International Conference on Software Engineering*, ICSE '11, 2011.

[43] Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny B. Sipma. Collecting statistics over runtime executions. *Form. Methods Syst. Des.*, 27(3):253–274, November 2005.

[44] Simos Gerasimou, Radu Calinescu, and Alec Banks. Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration. In *Proc. of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2014.

[45] Carlo Ghezzi, Mauro Pezzè, Michele Sama, and Giordano Tamburrelli. Mining behavior models from user-intensive web applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 277–287, New York, NY, USA, 2014. ACM.

[46] Carlo Ghezzi, Leandro Sales Pinto, Paola Spoletini, and Giordano Tamburrelli. Managing non-functional uncertainty via model-driven adaptivity. In *Proc. of the 2013 International Conference on Software Engineering*, ICSE '13, 2013.

[47] Carlo Ghezzi and Amir Molzam Sharifloo. Dealing with non-functional requirements for adaptive systems via dynamic software product-lines. In *Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010.* [1].

[48] Roberto Giacobazzi and Radhia Cousot, editors. *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. ACM, 2013.

[49] M. Glinz. On non-functional requirements. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pages 21–26, 2007.

[50] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction: A statistical learning approach. In *2013 28th IEEE/ACM*

International Conference on Automated Software Engineering (ASE), pages 301–311, Nov 2013.

[51] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang. A genetic algorithm for optimized feature selection with resource constraints in software product lines. *J. Syst. Softw.*, 84(12):2208–2221, December 2011.

[52] Svein Hallsteinsen, Erlend Stav, Arnor Solberg, and Jacqueline Floch. Using product line techniques to build adaptive systems. In *Proc. of the 10th International on Software Product Line Conference*, SPLC '06, 2006.

[53] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[54] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. Combining multi-objective search and constraint solving for configuring large software product lines. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 517–528, Piscataway, NJ, USA, 2015. IEEE Press.

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

[56] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Timed transition systems. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 226–251, Berlin, Heidelberg, 1992. Springer-Verlag.

[57] Thomas A. Henzinger and Jan Otop. From model checking to model measuring. In D'Argenio and Melgratti [34], pages 273–287.

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

[59] Robert M. Hierons, Miqing Li, Xiaohui Liu, Sergio Segura, and Wei Zheng. Sip: Optimal product selection from feature models using many-objective evolutionary optimization. *ACM Trans. Softw. Eng. Methodol.*, 25(2):17:1–17:39, April 2016.

[60] Mike Hinchey, Sooyong Park, and Klaus Schmid. Building dynamic software product lines. *Computer*, 45:22–26, 2012.

[61] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Inc., USA, 1990.

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

[63] Raj Jain. *The art of computer systems performance analysis*. Wiley, 1991.

[64] Jonas Finnemann Jensen, Kim Guldstrand Larsen, Jiří Srba, and Lars Kaerlund Oestergaard. Efficient model-checking of weighted ctl with upper-bound constraints. *Int. J. Softw. Tools Technol. Transf.*, 18(4):409–426, August 2016.

[65] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. Distance-based sampling of software configuration spaces. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1084–1094. IEEE / ACM, 2019.

[66] Kyo C Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis feasibility study. Technical report, SEI-CMU, 1990.

[67] Kyo C Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study. Technical report, Software Engineering Institute - CMU, 1990.

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

[69] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A variability-aware module system. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 773–792, New York, NY, USA, 2012. ACM.

[70] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[71] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 372–381, New York, NY, USA, 2005. ACM.

[72] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic model checking for performance and reliability analysis. *SIGMETRICS Perform. Eval. Rev.*, 36(4):40–45, March 2009.

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

[74] Jaejoon Lee and Kyo C. Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *Proceedings of the 10th International on Software Product Line Conference*, SPLC '06, pages 131–140, Washington, DC, USA, 2006. IEEE Computer Society.

[75] David Lo, Leonardo Mariani, and Pezze Mauro. Automatic steering of behavioral model inference. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 345–354, New York, NY, USA, 2009. ACM.

[76] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 501–510, New York, NY, USA, 2008. ACM.

[77] L. Mariani, M. Pezze, and M. Santoro. Gk-tail+ an efficient approach to learn software models. *IEEE Transactions on Software Engineering*, 43(8):738, 10 2017.

[78] Nicolas Markey and Philippe Schnoebelen. Symbolic model checking for simply-timed systems. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 102–117, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[79] G. A. Moreno, O. Strichman, S. Chaki, and R. Vaisman. Decision-making with cross-entropy for self-adaptation. In *12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, May 2017.

[80] Gabriel A. Moreno, Javier Camara, David Garlan, and Bradley Schmerl. Proactive self-adaptation under uncertainty: A probabilistic model checking approach. In *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, 2015.

[81] Gabriel A. Moreno, Javier Camara, David Garlan, and Bradley R. Schmerl. Efficient decision-making under uncertainty for proactive self-adaptation. In *2016 IEEE International Conference on Autonomic Computing*, 2016.

[82] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. Using bad learners to find good configurations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 257267, New York, NY, USA, 2017. Association for Computing Machinery.

[83] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. Using bad learners to find good configurations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 257–267, New York, NY, USA, 2017. ACM.

[84] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. Faster discovery of faster system configurations with spectral learning. *Automated Software Engineering*, 25(2):247–277, Jun 2018.

[85] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 61–71, New York, NY, USA, 2017. ACM.

[86] Tony Ohmann, Kevin Thai, Ivan Beschastnikh, and Yuriy Brun. Mining precise performance-aware behavioral models from existing instrumentation. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 484–487, New York, NY, USA, 2014. ACM.

[87] Rafael Olaechea, Joanne Atlee, Axel Legay, and Uli Fahrenberg. Trace checking for dynamic software product lines. In *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '18, pages 69–75, New York, NY, USA, 2018. ACM.

[88] Rafael Olaechea, Uli Fahrenberg, Joanne M. Atlee, and Axel Legay. Long-term average cost in featured transition systems. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 109–118, New York, NY, USA, 2016. ACM.

[89] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[90] Gustavo G. Pascual, Roberto E. Lopez-Herrejon, Mónica Pinto, Lidia Fuentes, and Alexander Egyed. Applying multiobjective evolutionary algorithms to dynamic software product lines for reconfiguring mobile applications. *J. Syst. Softw.*, 103(C):392–411, May 2015.

[91] Gustavo G. Pascual, Mónica Pinto, and Lidia Fuentes. Run-time adaptation of mobile applications using genetic algorithms. In *Proc. of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '13, 2013.

[92] F. Pastore, D. Micucci, and L. Mariani. Timed k-tail: Automatic inference of timed automata. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 401–411, March 2017.

[93] Malte Plath and Mark Ryan. Feature integration using a feature construct. *Sci. Comput. Program.*, 41(1):53–84, September 2001.

[94] Klaus Pohl, Gunter Bckle, and Frank J van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin, Heidelberg, 2005.

[95] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar. Scalable product line configuration: A straw to break the camel's back. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 465–474, Nov 2013.

[96] Micha Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers And Mathematics with Applications*, 7(1):67–72, 1981.

[97] Sol M. Shatz, Jia-Ping Wang, and Masanori Goto. Task allocation for maximizing reliability of distributed computer systems. *IEEE Trans. Computers*, 41(9):1156–1168, 1992.

[98] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kastner. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 284–294, New York, NY, USA, 2015. ACM.

[99] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kastner, Sven Apel, Don Batory, Marko Rosenmuller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 167177. IEEE Press, 2012.

[100] Fabio Somenzi. CUDD: CU Decision Diagram Package. https://add-lib.scce.info/assets/documents/cudd-manual.pdf, accessed 2021-09-09.

[101] M. H. ter Beek, A. Legay, A. Lluch Lafuente, and A. Vandin. Statistical analysis of probabilistic models of software product lines with quantitative constraints. In *Proc. of the 19th International Conference on Software Product Line*, SPLC '15, 2015.

[102] Thomas Thum, Sven Apel, Christian Kastner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, June 2014.

[103] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the royal statistical society series b-methodological*, 58:267–288, 1996.

[104] Andrey N. Tikhonov and Vasiliy Y. Arsenin. *Solutions of ill-posed problems*. V. H. Winston & Sons, Washington, D.C.: John Wiley & Sons, New York, 1977. Translated from the Russian, Preface by translation editor Fritz John, Scripta Series in Mathematics.

[105] Mahsa Varshosaz and Ramtin Khosravi. Discrete time markov chain families: Modeling and verification of probabilistic software product lines. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, SPLC '13 Workshops, pages 34–41, New York, NY, USA, 2013. ACM.

[106] Danny Weyns and Radu Calinescu. Tele assistance: A self-adaptive service-based system examplar. In *Proc. of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '15, 2015.

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