Trace Checking for Dynamic Software Product Lines

Rafael Olaechea*, Joanne M. Atlee*, Axel Legay**, Uli Fahrenberg***
* University of Waterloo, Canada; Email: {reolaech, jmatlee}@uwaterloo.ca
** INRIA - Reness, France; Email: axel.legay@irisa.fr
*** Ecole Polytechnique, Palaiseau, France; Email: ulrich.fahrenberg@polytechnique.edu

ABSTRACT

A key objective of self-adaptive systems is to continue to provide optimal quality of service when the environment changes. A dynamic software product line (DSPL) can benefit from knowing how its various *product variants* would have performed (in terms of quality of service) with respect to the recent history of inputs. We propose a family-based analysis that simulates all the product variants of a DSPL simultaneously, at runtime, on recent environmental inputs to obtain an estimate of the quality of service that each one of the product variants would have had, provided it had been executing. We assessed the efficiency of our DSPL analysis compared to the efficiency of analyzing each product individually on three case studies. We obtained mixed results due to the explosion of quality-of-service values for the product variants of a DSPL. After introducing a simple data abstraction on the values of quality-of-service variables, our DSPL analysis is between 1.4 and 7.7 times faster than analyzing the products one at a time.

1. INTRODUCTION

An important objective of reconfiguration of a self-adaptive system is to continually provide optimal quality of service while operating in a changing environment. We study this problem in the context of a dynamic software product lines (DSPL), which are a type of self-adaptive system that have a set of optional features — units of functionality — that can be activated or deactivated at runtime. A DSPL has a large but fixed number of *product variants* or *configurations*, which are defined by the set of optional features that are active. 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 what the target configuration should be.

Existing approaches maintain and update a model of the environment that a self-adaptive system encounters 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 [2, 18]. 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 [9, 19], by performing an approximate analysis that returns the best configuration found (so far) after a fixed amount of time has elapsed [20], and by identifying near-optimal configurations [12]. In contrast, trace checking offers an opportunity to very quickly estimate how each configuration of a self-adaptive system would have performed over the recent inputs. Alternatively, other approaches use coarse-grained models that estimate the impact that each adaptation [3] or each activation/deactivation of a feature [21] 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 a self-adaptive systems nor the interactions between multiple features.

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 in the observed execution. Trace checking has been applied to estimate the quality of service provided by a single software system in an observed execution trace [11]. However, the number of configurations of a self-adaptive system increases exponentially with the number of optional features. Therefore, we propose a *family-based* analysis that analyzes all the configurations simultaneously to speed up the analysis.

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 our family-based analysis that tries to take advantage of commonalities in the analysis results among configurations. Thus, we introduce a simple data abstraction over the values of quality attributes to facilitate sharing of partial analysis results across different

SEAMS '18 May 28–29, 2018, Gothenburg, Sweden

ISBN 978-1-4503-5715-9/18/05…$15.00

DOI: 10.1145/3194133.3194143

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 analyses of each configuration individually in all case studies.

The main contributions of this paper are:

- A family-based trace-checking algorithm (and its implementation) that analyzes the quality of service that each configuration of a DSPL would have provided over recent system inputs.
- An evaluation of the efficiency of such analysis on three DSPLs 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 paper is organized as follows. In Section 2 we introduce an example DSPL that is used as a running example throughout the rest of the paper, in Section 3 we review background formalisms, and in Section 4 we present our family-based trace checking algorithm. In Section 5 we present the results of our evaluations and discuss our findings. In Section 6 we describe related work, and in Section 7 we present our conclusions.

2. RUNNING EXAMPLE

As a running example, consider 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. 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 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.

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 keeping the additional motor idle.

Thus the ideal configuration of the DSPL varies with the conditions of the the environment.

3. BACKGROUND

In this section we provide an overview of dynamic software product lines (DSPLs), techniques for modelling the behaviour and quality of service provided by a DSPL, and tools for analyzing such models. Specifically, we review transition systems, featured transition systems, weighted featured transition systems, and trace checking.

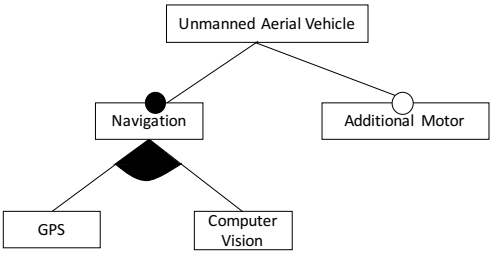

Figure 1: Feature model for an unmanned aerial vehicle dynamic software product line.

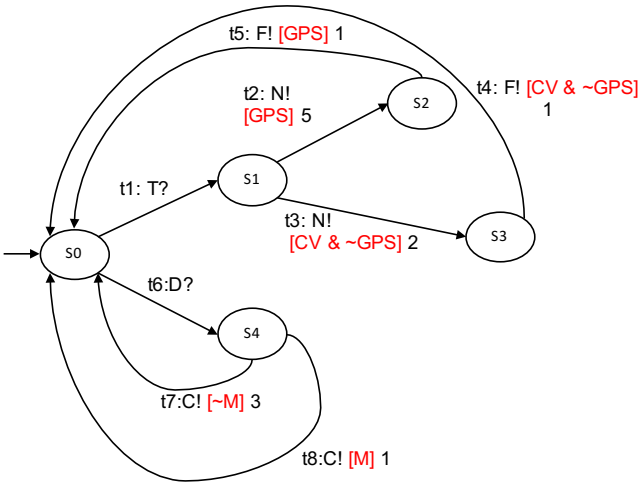

Figure 2: A weighted featured transition system for the unmanned aerial vehicle DSPL.

3.1 Dynamic Software Product Lines

A *dynamic software product line* (DSPL) [15] is a type of self-adaptive system that leverages tools and techniques from software product-line (SPL) engineering to facilitate self-adaptation. In the context of DSPLs, a *feature* is a coherent unit of user-visible functionality. A feature can be either mandatory or optional. A specific software product variant, called *configuration*, comprises of mandatory features and a selected subset of optional features.

Not all combinations of features are valid software configurations. A *feature model* [16] 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 1 shows a feature model for our example unmanned aerial vehicle DSPL. 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, GPS or Computer Vision, must be selected in any product that includes feature Navigation.

DEFINITION 1. *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 *representing all valid configurations.*

In traditional software product-line engineering, variability is resolved at design time by selecting which of an SPL's optional features to include in the product being derived. In contrast, in a dynamic software product-line, variability can be resolved and can change at runtime by dynamically activating or deactivating features. Thus, a DSPL can respond to changes in its execution environment by modifying its active configuration at runtime.

3.2 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.

DEFINITION 2. *A transition system (TS) is a tuple* $ts = (S, A, I, T)$*, where* S *is a set of states,* A *is a set of actions,* $I \subseteq S$ *is a set of initial states, and* $T \subseteq S \times A \times S$ *is a set of transitions.*

Each action is labelled as either a system action (!) or an environment action (?). An *execution* π of a transition system is an alternating infinite 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.

A transition system models the behaviour of an individual software product, whereas a software product-line (including a DSPL) has many product variants, or configurations. Classen et al. [6] introduce *Featured Transition Systems* (FTSs) to model an SPL and the behaviour of all its configurations in a single model. Each optional feature is represented by a boolean *feature variable*, whose value denotes whether the feature is activated (true) or deactivated (false). Thus, a software configuration can be represented by an assignment of values to all the feature variables. In an FTS, each transition is annotated with a *feature expression* — a propositional formula ranging over feature variables — that denotes the set of configurations that exhibit that transition. For example, the feature expression $GPS \wedge CV$ denotes all the configurations that include both feature GPS and feature CV. The set of all propositional formulas ranging over feature variables is denoted $\mathbb{B}(N)$. More formally:

DEFINITION 3. *An FTS is a tuple* $fts = (S, A, I, T, d, \gamma)$*, where* (S, A, I, T) *is a transition system,* $d = (N, px)$ *is a feature model, and* $\gamma : T \to \mathbb{B}(N)$ *labels each transition with a feature expression.*

A transition system for a specific software configuration can then be derived from an FTS model by including only the transitions whose feature expressions are satisfied by the configuration's feature-variable assignment. For example, a configuration of the UAV that includes only the feature GPS would exhibit only transitions t1, t2, t5, t6, and t7.

To model the quality aspects of a system, transition systems (and featured transition systems) have been extended with *weights* [1]. 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 a DSPL, whereas a *weighted featured transition system* (WFTS) models the quality of service on all transitions in all configurations of a DSPL. More formally:

DEFINITION 4. *A weighted featured transition system (WFTS) is a tuple* $wfts = (S, A, I, T, d, \gamma, W)$*, where* (S, A, T, I, d, γ) *is an FTS and* $W : T \to \mathbb{R}$ *is a function that annotates each transition with a weight value.*

3.3 Trace Checking

Trace checking [8, 11] is a lightweight formal-analysis method that analyzes whether an observed execution trace of a system satisfies its 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 [11]. 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. 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.

4. 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. We also extend this approach with data abstraction to improve its runtime performance.

4.1 Family-based Trace Checking

Different configurations will react to environmental inputs 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 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 configuration on a sequence of environmental actions. 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 M records triplets of feature expressions, system state, and accumulated reward. The algorithm updates the data structure as actions from the execution trace are processed. It refines (splits) a tuple when

Alg. 1: Family-based simulation of a DSPL

```
   Procedure Simulate-FTS-Execution(Trace)
1  Input: Trace = s0α0s1, s1α1s2 ...
2  begin
3    M ← {(s0, ⊤, 0)}
4    for each st = si, αi, si+1 ∈ Trace
5      M′ ← {}
6      if IsEnvironmentAction (α):
7        for each (s, γ, r) ∈ M, t =(s, β, sdst) ∈ T, αi = β
8          MergeTriplets(M′, (sdst,γ ∧ γ(t), r + W(t)))
9      else:
10       for each (s, γ, r) ∈ M, t =(s, β, sdst) ∈ T
11       s.t. γ ∧ γ(t) ⊭ ⊥ and IsSystemAction(β)
12         MergeTriplets(M′, (sdst, γ ∧ γ(t), r + W(t)))
13     M ← M′
14 end
   Procedure MergeTriplets (M′, (s, γ, r))
15 begin
16     if ∃ψ s.t. (s, ψ, r) ∈ M′
17       M' ← M' \{(s, ψ, r)}
18       M' ← M' ∪{(s, γ ∨ ψ, r)}
19     else
20       M' ← M' ∪{(s, γ, r)}
21 end
```

different configurations in its feature expression would result in different accumulated rewards or system state. For example, a tuple with feature expression $\top$ (representing all configurations), system state s_1, 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 M to contain a single triplet consisting of the feature expression $\top$ (representing all configurations), the initial state, and an accumulated reward of zero (line 3). 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 6), the algorithm determines the next state of execution of each configuration as a result of executing the transitions triggered by the configurations' current state and the trace's environment action. For each tuple $< s, \gamma, r >$ in M and for each transition t triggered by action α (line 7), 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 $\gamma(t)$); and accumulated reward $r + W(t)$. For example, if M 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 1-8, 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. Specifically, when the current configuration executes a system action α (line 9), our algorithm, for each tuple $< s, \gamma, r >$ in M (line 10), considers any transition t that has a matching source state s and whose feature expression $\gamma(t)$ is compatible with the tuple's feature expression (γ) (lines 10-12), to generate a new tuple (line 12).

After processing an action, the resulting triplets are collected into a new version of M, denoted M'. The procedure *MergeTriplets* is responsible for updating M' 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 M (ψ) to a new feature expression that is the disjunction of ψ and the new feature expression γ (lines 17-18). If the pair s and r doesn't already exist in some tuple in M', then the algorithm updates M' to include the new tuple (s, γ, r) (line 20).

Consider Figure 2, which shows a weighted featured transition system for our running example unmanned air vehicle DSPL. 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. Consider the execution trace s_0,T?,s_1,N!,s_2,F!,s_0 for the configuration exclusively contains 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 M 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. For each, the algorithm computes a tuple of feature expression, a resulting system state, and an 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 M comprises triplets $< GPS, 1, 6 >$ and $< CV \wedge \neg GPS, 1, 3 >$. These result 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

Table 1: Average time taken by product-based, family-based, and abstract family-based trace checking.

System	# of features	# of configurations	# of states	Product-based analysis (s)[1]	Family-based analysis (s)[1]	Family-based analysis with abstraction (s)[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 ± std. dev. The times for

ProVeLines [7], which is a family-based model checker for software product line models. We 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 simulates executing each product over the observed trace.

4.2 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, which prevent sharing of partial analysis results across those products. However, for some systems, it may be acceptable to disregard small differences when making reconfiguration decisions. We use data abstraction on the value of transition weights to obtain a faster 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 transitions that each set of configurations would have executed. Our algorithm is then able to group two configurations together if they would have executed the same number of high-weight transitions.

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.

5. 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 (with and without data abstraction) against a product-based analysis that analyzes each configuration individually. We obtain mixed results for the performance of our family-based algorithm without data abstraction, whereas with data abstraction our family-based algorithm is between 1.4 and 7.7 times faster than the product-based approach in all case studies. We discuss hypotheses for why our family-based analyses performs better on some systems than on others.

5.1 Subject Systems

The first case study is a tele-assistance system (taken from [25]). 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. For data abstraction, we manually partition services (represented by transitions) into those with a high cost and those with a low cost. We manually translated a model of this system into a version of Promela that is extended with support for features [6]. This system has six features that conform 9 configuration.

A second case study is an e-commerce application taken from [14]. 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 users location. Each implementation of a feature is annotated with the amount of time it would take to execute it. For data abstraction, we manually classified features (represented as transitions) into those that were fast and those that were slow. We manually translated a model of this system into a version of Promela that is extended with support for features [6]. This application has seven features that conform 24 configurations.

A third case study is a model of an elevator system developed by Plath and Ryan [23] and extended by Classen et al. [5]. 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. We set the cost of moving the elevator one floor to one unit, so we do not use any data abstraction in this case study. The elevator model has eight features and 256 configurations.

5.2 Experimental Setup

We generated 20 random traces for each subject system. We set the length of the traces at 20000 transitions for the tele-assistance system and at 150000 transitions for the e-commerce application. We executed our family-based algorithm, with and without data abstraction, and the product-based approach 20 times on each trace and recorded the time taken by each execution.

The elevator system has much more nondeterminism than the other two systems, so our trace checking has to keep track of a much larger number of possible system states and

takes a longer time to analyze the system. Thus in the elevator case study, we used a much smaller trace that consists of consisting of 10 target floor choices (for each of the two passengers) — that is, our trace records the floors that each of the two passengers will request when they enter the elevator.

5.3 Results and Discussion

Table 1 shows the average and standard deviation for the time taken by our family-based analysis, with and without data abstraction, and by the product-based analysis for each one of the three subject systems.

Our family-based trace-checking analysis without data abstraction is 5.7 times faster than the product-based analysis on the elevator system, it 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., [5, 6]). 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). Thus, we think 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. Thus, most configurations would likely generate different accumulated rewards for those two systems in the analyzed traces. 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 why our family-based algorithm obtains a large speed-up on the elevator system, but no or marginal speed-ups on the other two systems. We hypothesize that our analysis, without data abstraction, will perform well on systems that have only a few different values for their transition's weights.

In contrast, our family-based algorithm, with data abstraction, performs much better and is faster than the product-based analysis for all three case studies. We observe that such algorithm is 1.39 times faster than the product-based approach 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).

6. RELATED WORK

A class of approaches [4, 21, 22] associate with each feature or adaptation a static value for quality of service and a static value for cost or resource usage. These values apply (or not) depending on whether (or not) the feature is active in a configuration. Approach [21, 22] 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. Approach [24] introduces a language to model the probabilities and constraints of activating/deactivating features at runtime. 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). This allows for a more accurate assessment of a system's quality that takes into account not only changes of the operating environment but also changes in the executing system.

In approaches that use rewarded Discrete-Time Markov Chain (DTMC) models [2, 10, 13], 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. In [2], a self-adaptive system 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. [12] improves such analysis performance by caching previously seen configurations and preemptively analyzing configurations that will likely be encountered. To reduce runtime analysis time, [9, 10] 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. [13] 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 [9] 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. In contrast, our work does not assume feature to be independent and permits transitions to depend on multiple features.

Proactive adaptation [18] 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 [17] is used to decide which adaptation maximizes the overall utility over a small look-ahead period. For scalability, [20] adapts an approximate optimization algorithm to find near-optimal configurations much faster. 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.

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 is substantially faster than analyzing each configuration individually in only one system. 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.

In future work, we plan to study specialized data abstractions for different quality attributes, including the unique ways in which a specific type of quality attribute evolves over time. This way we expect to ease the task of identifying suitable abstractions for quality attributes.

References

[1] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[2] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimization in service-based systems. *IEEE Trans. Softw. Eng.*, 37(3):387–409, 2011.

[3] J. Camara, A. Lopes, D. Garlan, and B. R. Schmerl. Impact models for architecture-based self-adaptive systems. In *Formal Aspects of Component Software - 11th International Symposium, FACS 2014*, 2014.

[4] S.-W. Cheng and D. Garlan. Stitch: A language for architecture-based self-adaptation. *J. Syst. Softw.*, 85 (12):2860–2875, Dec. 2012.

[5] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *Proc. of the 33rd International Conference on Software Engineering*, ICSE '11, 2011.

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

[7] M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Provelines: A product line of verifiers for software product lines. In *Proc. of the 17th International Software Product Line Conference, SPLC*, 2013.

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

[9] A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In *Proc. of the 33rd International Conference on Software Engineering*, ICSE '11, 2011.

[10] 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.

[11] B. Finkbeiner, S. Sankaranarayanan, and H. B. Sipma. Collecting statistics over runtime executions. *Form. Methods Syst. Des.*, 27(3):253–274, Nov. 2005.

[12] S. Gerasimou, R. Calinescu, and A. 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.

[13] C. Ghezzi and A. M. 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.*, 2010.

[14] C. Ghezzi, L. S. Pinto, P. Spoletini, and G. Tamburrelli. Managing non-functional uncertainty via model-driven adaptivity. In *Proc. of the 2013 International Conference on Software Engineering*, ICSE '13, 2013.

[15] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using product line techniques to build adaptive systems. In *Proc. of the 10th International on Software Product Line Conference*, SPLC '06, 2006.

[16] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis feasibility study. Technical report, SEI-CMU, 1990.

[17] M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic model checking for performance and reliability analysis. *SIGMETRICS Perform. Eval. Rev.*, 36 (4):40–45, Mar. 2009.

[18] G. A. Moreno, J. Camara, D. Garlan, and B. 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.

[19] G. A. Moreno, J. Camara, D. Garlan, and B. R. Schmerl. Efficient decision-making under uncertainty for proactive self-adaptation. In *2016 IEEE International Conference on Autonomic Computing*, 2016.

[20] 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.

[21] G. G. Pascual, M. Pinto, and L. 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.

[22] G. G. Pascual, R. E. Lopez-Herrejon, M. Pinto, L. Fuentes, and A. Egyed. Applying multiobjective evolutionary algorithms to dynamic software product lines for reconfiguring mobile applications. *J. Syst. Softw.*, 103(C):392–411, May 2015.

[23] M. Plath and M. Ryan. Feature integration using a feature construct. *Sci. Comput. Program.*, 41(1):53–84, Sept. 2001.

[24] M. H. ter Beek, A. Legay, A. L. 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.

[25] D. Weyns and R. 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.