Extending Dynamic Software Product Lines with Temporal Constraints

Gustavo Sousa, Walter Rudametkin, Laurence Duchien

Extending Dynamic Software Product Lines with Temporal Constraints

Gustavo Sousa, Walter Rudametkin, Laurence Duchien
University of Lille / Inria
Lille, France
firstname.lastname@inria.fr

***Abstract*—Due to the number of cloud providers, as well as the extensive collection of services, cloud computing provides very flexible environments, where resources and services can be provisioned and released on demand. However, reconfiguration and adaptation mechanisms in cloud environments are very heterogeneous and often exhibit complex constraints. For example, when reconfiguring a cloud system, a set of available services may be dependent on previous choices, or there may be alternative ways of adapting the system, with different impacts on performance, costs or reconfiguration time.**

Cloud computing systems exhibit high levels of variability, making dynamic software product lines (DSPLs) a promising approach for managing them. However, in DSPL approaches, verification is often limited to verifying conformance to a variability model, but this is insufficient to verify complex reconfiguration constraints that exist in cloud computing systems.

In this paper, we propose the use of temporal constraints and reconfiguration operations to model a DSPL's reconfiguration lifecycle. We demonstrate how these concepts can be used to model the variability of cloud systems, and we use our approach to identify reconfigurations that meet given criteria.

***Keywords*-variability; dynamic software product lines; feature models; cloud computing;**

I. Introduction

In the cloud computing paradigm, computing resources are delivered to customers as services at different levels of abstraction, such as infrastructure, platform and software [1]. Each cloud computing provider offers a different set of services, such as processing power, network communication, virtual machines, containers, software packages, application servers, databases, development and management tools, etc. When deploying an application to the cloud, customers can select a set of services to build a cloud environment that supports their applications' requirements.

However, cloud providers are very heterogeneous, and each provider may have a different set of complex rules governing the selection of services. To deal with this complexity, recent works [2], [3], [4], [5] have proposed the use of software product line (SPL) techniques, such as feature models, to capture the variability of cloud configurations, allowing the automated setup of cloud environments. The cloud computing model also supports rapid provisioning and release of resources [1], providing mechanisms for reconfiguring an existing environment to cope with new application requirements or context changes. The high variability in cloud configuration, together with the support for dynamic provisioning, makes dynamic software product lines (DSPLs) a promising approach to build adaptive cloud environments.

However, mechanisms offered for adapting cloud environments may differ substantially between providers, and complex constraints may also apply to reconfigurations. Depending on providers, the same context or requirements change may require a different set of reconfiguration operations. Even within the same provider, multiple alternative ways of performing the same reconfiguration may be offered, each one with a different impact on performance, costs and downtime. Additionally, choices made in the initial or previous configurations may affect the set of available future reconfigurations, thus changing the system's variability over its lifecycle. In a sense, reconfigurations in cloud systems are not stateless, they depend on the system's current and past configurations, but this information is not captured in DSPLs.

Previous work on DSPLs [6], [7] highlight the importance of guaranteeing a safe transition from a valid initial configuration to a target configuration. However, in most DSPL approaches verification is restricted to checking if a target configuration complies to its variability model. In existing approaches, there is no regard about how the system transitions between configurations, if there are any constraints over these transitions, or if multiple alternatives are available.

In this paper, we propose extending feature models with temporal constraints and reconfiguration operations to model the variability of DSPLs. We introduce reconfiguration operations to enable describing multiple alternative reconfiguration paths, together with their associated costs. We demonstrate how principles from model checking can be used to implement reasoning over these concepts, allowing us to check for reconfiguration request validity and to find reconfigurations that meet a specific criteria (e.g., downtime, costs). Finally, we evaluate our approach with a use case on adaptive cloud computing environments.

In Section II, we discuss the motivation for using temporal constraints in dynamic software product lines for cloud computing. In Section III, we define the semantics of temporal logic constraints and reconfiguration operations on feature models. Section IV describes the implementation of a tool for automated reasoning over feature models with temporal constraints and reconfiguration operations. Finally, we discuss related work in Section VI and the conclusions in Section VII.

II. MOTIVATION

When a cloud environment is initially setup for an application, it is conceived to support initial application requirements. However, as application requirements evolve or the system load changes, its environment should be reconfigured to deal with these new requirements. These changes may include the provisioning of extra resources such as virtual machines or application containers when more requests arrive, or upgrading to a larger database plan as the system starts having more users, or changing the application framework to cope with application evolution.

Systems that exhibit runtime variability as the cloud environments are candidates for employing a DSPL architecture. DSPLs leverage concepts and engineering foundations from SPL to support the construction of adaptive systems [8]. The core element of both approaches is a variability model that specifies variation points in which members of a product line can differ. Feature models are widely used to model variability in product line approaches, and many methods for their automated analysis are available [9].

In DSPL approaches, feature models may be enriched with context [10] and binding time [11], [12], [13] information. A reconfiguration is usually triggered by monitored context features or another system component requesting the inclusion (exclusion) of a set of features to (from) the current configuration. Techniques for automated analysis of feature models are then used to verify if the requested configuration is valid according to feature model structure and constraints. Once a target configuration is validated, adaptation mechanisms from the underlying platform (e.g. component models, reflection, models@runtime) are used to enact the it.

However in cloud computing, reconfiguration mechanisms may vary according to the provider, and even for the same provider different services may have different reconfiguration support. While some services may have their settings updated at runtime, others may require a service restart or still a complete redeployment of the service.

A. Motivating example

To illustrate the problems faced while managing variability and adaption in cloud environments we present an example from the Heroku PaaS provider. The feature model in Fig. 1 describes part of the variability in the configuration of Heroku PaaS. In Heroku, users can define application projects that include support for development in a given programming language such as `Java`, `Ruby` or `PHP` and a set of additional cloud services such as databases, caching, monitoring tools, etc. The cloud customer can choose in which geographical region (`US` or `EU`) he wants the application project to be deployed and cloud services may be available at different plans (`M1`, `M2`, `H1`, `H2`, `S1`, `S2`, etc), with different capabilities and costs.

While initially setting up an application project, customers have access to all cloud services, and can select any configuration that complies with the feature model. However, after the initial setup, reconfigurations may be limited by the initial

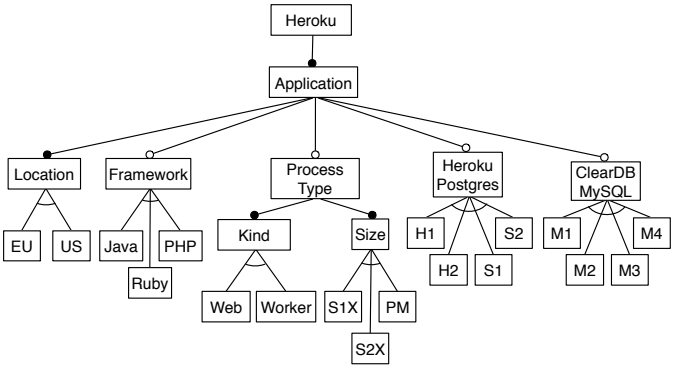

Fig. 1. Partial feature model for the Heroku PaaS provider

choices or require extra operations. Below we present some of the reconfiguration constraints identified on the Heroku PaaS provider.

- *MySQL database plan.* Four different plans are available for the ClearDB MySQL service, each one with different storage and pricing. Heroku provides support for upgrading directly from a smaller to a larger plan[1], however it does support downgrading to a smaller plan. Thus, once a choice is made, future reconfigurations should not consider smaller plans, even if the feature model describe it as a possible alternative.
- *PostgreSQL database plan.* Heroku Postgres provides a wide range of service plans classified into three categories: Hobby, Standard and Premium. Heroku proposes two ways to upgrade or downgrade between different plans[2]. The first way relies on the use of `pgcopy` tool, and requires a downtime of about 3 minutes per GB. The second method creates a follower database with the new desired plan and wait until it to syncs with the original database. This process can take several hours, in which additional costs for maintaining two databases should be considered, but downtime is minimal or unnoticeable. Reconfigurations from or to Hobby plans are only possible with the `pgcopy` method.
- *Framework type.* Changing the framework[3] (buildpack) can be achieved by updating a configuration variable and redeploying the application code. The executing instances of the application server are restarted.
- *Application container size.* Application container instances (called dynos in Heroku) can be easily resized at runtime[4] by resetting the size attribute. Heroku platform will take care of alternately replacing instances of previous size by new instances of the requested size.
- *Application location.* Changing the location of an appli-

[1] https://devcenter.heroku.com/articles/cleardb#upgrading-your-cleardb-database

[2] https://devcenter.heroku.com/articles/upgrading-heroku-postgres-databases

[3] https://devcenter.heroku.com/articles/buildpacks#setting-a-buildpack-on-an-application

[4] https://devcenter.heroku.com/articles/scaling#scaling-the-dyno-size

cation project requires migrating it to another data center[5] and therefore implies a series of complex operations. This may include the migration of application code and data. Similar to the examples above, this can be achieved in different ways by combining different operations, with different effects on costs, downtime and performance of the application during migration.

Even from this simple example, we can notice that cloud environments may be subject to complex constraints governing when and how a new configuration can be reached. Existing DSPL approaches do not provide enough support for modeling or reasoning over reconfigurations constraints and are restricted to verification of the structural variability.

B. Challenges

We wish to provide support for automated reasoning on reconfigurations in cloud systems. To do so, we require modeling the constraints and reconfiguration operations of a cloud system in a way that allows for a generically, independent of cloud providers. In this paper, we investigate how reconfiguration constraints can be added to feature models to support modeling and reasoning over cloud environments adaptation. More specifically, we try to tackle the following challenges.

- *Model temporal constraints on DSPLs*. How to define constraints that include temporal aspects, such as when selecting a feature makes another feature unavailable for all future reconfigurations?
- *Model reconfiguration operations and constraints* How to capture multiple alternative reconfiguration paths and their impact on performance, costs and reconfiguration time?
- *Finding reconfigurations* How to implement reasoning over feature models and reconfiguration constraints to answer queries such as 'what is the cheapest reconfiguration path to meet a set of requirements' or 'is there a reconfiguration path in which there is no downtime?'.

III. Modeling DSPL adaptation

Previous works [7], [6] on DSPL highlight the need for adaptive systems to perform safe transitions between configurations. According to *Morin et al.* [6], systems should evolve through a safe migration path between configurations, while *Hubaux et al.* [7] say that transitions should not only consider static constraints of a variability model but also the "dynamic constraints that determine the allowed transitions between configurations".

In spite of it, existing works on DSPLs provide little support for reasoning over transitions between configurations. In most approaches, verification is limited to checking if a target configuration conforms to a variability model.

In some approaches, the variability model is augmented with context features [14], [15], [16], [10] or binding time [11], [13] information. Still, verification is restricted to monitoring context changes, checking what features need to be (de)selected and verifying if these features can be rebound at runtime.

[5]https://devcenter.heroku.com/articles/app-migration

According to [17], [6], [18] a DSPL can be abstracted as a reactive transition system [19] in which states represent valid system configurations and state transitions define system adaptations. The variability model is therefore a compact notation for describing a DSPL's transition system. By using a variability model we eliminate the need to enumerate all possible system configurations and transitions.

However, as a variability model abstracts away from its underlying transition system, it fails to capture any restriction that may exist on the transitions between configurations. By introducing temporal constraints and reconfiguration operations we give first class status to these transitions while looking for a compromise between full specification of a transition system and the more abstract view based on variability models.

A. Preliminary definitions

In the following paragraphs, we will go over definitions for feature models, transition systems and DSPLs. Then, we will introduce temporal constraints and reconfiguration operations to DSPLs.

Definition 1: A feature model can be defined as a tuple $M = (\mathcal{F}, P)$ where $\mathcal{F}$ is the set of features and $P \subseteq \mathcal{P}(\mathcal{F})$ is the set of valid products that can be derived from the feature model. We also use the $[\![M]\!]_{FM}$ notation to denote the set of products of a feature model M.

Definition 2: A transition system [19] can be described by a Kripke structure [20] $T = (S, I, R, AP, L)$ where S is a finite set of system states, $I \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a left-total relation representing state transitions, AP is a set of propositions and $L : S \to 2^{AP}$ is a labeling function. An execution path of T is a non-empty infinite sequence of states $\rho = \langle s_0, s_1, s_2, s_3, ... \rangle$ such that $s_0 \in I$ and $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. The behavior of a transition system t is defined by the set all possible execution paths, denoted by $[[T]]_{TS} \subseteq S^\omega$ where S^ω is the set of all infinite sequences of states in T.

Definition 3: Given a feature model $M = (\mathcal{F}, P)$, the corresponding DSPL can be represented as a transition system $DSPL(M) = (S, I, R, AP, L)$ where $I = S = P$, $R = S \times S$, $AP = \mathcal{F}$ and $L(s) = s$.

From this definition, a DSPL can be seen as transition system in which each state represents a possible configuration according to the feature model. The system can start at any configuration and transition to any other possible configuration.

B. Temporal constraints

A temporal property [21] defines a condition over the execution paths of a transition system. A system is said to satisfy a given property if all of its possible execution paths satisfy the property. Thus, a temporal property specifies the admissible or desired behavior of a system over time.

Definition 4: Let $T = (S, I, R, AP, L)$ be a transition system, a temporal property $P \subseteq S^\omega$ over T is a set of execution paths over the states of T. The transition system T satisfies the property P, written $T \models P$ iff $[\![T]\!]_{TS} \subseteq P$.

Temporal properties are commonly used in model checking [19] to verify if a system exhibits a certain behavior, but can also be used to specify how the system should behave. Adding temporal properties to DSPL specification, allows to constrain the transitions between configurations to only those that satisfy the property. Thus, by combining feature models with temporal constraints we can define more precisely the adaptive behavior of a DSPL.

The semantics of a transition system and temporal properties are defined by sets of execution paths over system states. Transition systems and properties can be treated as languages over the set of system states. The execution paths are infinite words over the alphabet $\Sigma = S$. Thus, given a transition system TS, we say $\mathcal{L}(TS)$ is the language that represents all possible executions of TS. Similarly, for a property P, $\mathcal{L}(P)$ is the language that represents all the possible executions that satisfy P. The intersection of $\mathcal{L}(TS) \cap \mathcal{L}(P)$ is the set of execution paths of TS that satisfy P.

Therefore, given a feature model M and a set of properties $\phi \subseteq \mathcal{P}(S^\omega)$, we want to build a transition system TS_{DSPL} such that

$$\mathcal{L}(TS_{DSPL}) = \bigcap_{x \in \phi} \mathcal{L}(DSPL(M)) \cap \mathcal{L}(x).$$

That is a transition system whose execution paths are infinite sequences of configurations for feature model M that conform to all temporal properties ϕ.

Linear temporal logic (LTL) [19] is one of the most commonly used formalisms for expressing temporal properties and can represent a subset of the class of ω-regular properties. ω-regular properties correspond to the class of ω-regular languages, an extension of regular languages to infinite words.

In our work, we employ LTL as a formalism for defining temporal constraints, but other notations [19] could also be used applying the same principles. In LTL, formulas are composed of a finite set of atomic propositions AP, boolean connectors such as $\neg$, $\wedge$, $\vee$, $\rightarrow$, and temporal logical operators $\mathcal{U}$ (until), $\bigcirc$ (next time), $\Box$ (always) and $\diamond$ (eventually). Given $p \in AP$, the syntax of an LTL formula can be defined by the following grammar:

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid$$
$$\phi \,\mathcal{U} \phi \mid \bigcirc \phi \mid \Box\phi \mid \diamond \phi$$

Besides the standard boolean logic operators, LTL temporal operators can be defined in terms of the $\bigcirc$ (next) and $\mathcal{U}$ (until) operators. The intuitive semantics of these operators is that $\bigcirc\phi$ means that proposition ϕ should hold in the next state of a path. Thus the formula $\phi \wedge \bigcirc\psi$ defines the ϕ should hold in the current state and ψ should hold in the next state. The until operator defines that a a proposition should hold until another becomes true. Therefore $\phi \,\mathcal{U}\, \psi$ means that ϕ should hold from now on up to when ψ becomes true.

When defining temporal constraints over feature models the set of atomic propositions is defined by the set of features. Thus, an atomic proposition is true at a given state if the corresponding feature is selected in the corresponding configuration.

Using LTL we can express some constraints shown in the motivating example in Section II-A. For example, in the case of constraints on downgrading MySQL database plan, we could use the following LTL formula to define that a configuration using plan $M2$ cannot be downgraded to use $M1$.

$$\Box(M2 \rightarrow \neg \diamond M1)$$

Actually, this constraint expresses that if at any point in time ($\Box$), if the system goes though a state in which $M2$ is selected, then $M1$ should not eventually ($\neg\diamond$) be selected in the future.

As part of the class of ω-regular languages, LTL formulas are closed under union and intersection. Thus, given LTL formulas ϕ and ψ, $\mathcal{L}(\phi) \cap \mathcal{L}(\psi) = \mathcal{L}(\phi \ \wedge \ \psi)$.

From an LTL formula ϕ, we can build a corresponding Büchi automaton $\mathcal{A}_\phi$ that accepts exactly the infinite sequences of states that comply with ϕ. Büchi automata are an extension of finite automata for ω-regular languages and thus can be composed with a transition system to obtain another transition system that represents the intersection of both languages.

Thus, given a feature model and a temporal constraint, we can define a DSPL transition system whose execution paths are sequences of feature model configurations that conform with the temporal constraint.

Definition 5: Given a feature model M and a LTL formula ϕ, we can build DSPL with temporal constraints as $DSPL(M, \phi) = DSPL(M) \otimes \mathcal{A}_\phi$. That is, the synchronized product between the transition system for the DSPL defined by M, and the Büchi automata for ϕ.

As discussed earlier, a feature model M defines a corresponding transition system $DSPL(M)$ in which there are transitions between any two possible feature model configurations. The goal of including temporal constraints is to limit these transitions and express the actual variability of the system over time.

By combining a feature model with temporal constraints we can define a DSPL's adaptive behavior in a compact way. Temporal constraints can be defined using well-known LTL logical formalism, which can be transformed into a Büchi automaton and combined with the feature model into a transition system.

C. Reconfiguration operations

In the previous section we introduced temporal constraints to feature models to define properties over a DSPL's adaptive behavior. For the analysis of temporal properties, we did not consider any actions or labeling of transitions and analysis was strictly state-based.

However, as seen in the motivating example, reconfiguring a cloud environment may require executing some operations such as migrating a database, restarting services, redeploying

an application project, etc. Each of these operations may have different impacts on the application cost and performance, and therefore should be considered during the reconfiguration process.

For modeling a DSPL with reconfiguration operations we rely on the definition of doubly labeled transition systems (L^2TS) [22], in which both states and actions can be labeled with atomic propositions. In the literature, other similar notations such as labeled Kripke structures [23] and mixed transition systems [24] have also been used for state/action-based analysis.

Definition 6: A L^2TS is a tuple $T = (S, I, Act, R, AP, L)$ where S is a finite set of system states, I is the set of initial states, Act is a finite set of transition actions, $R \subseteq S \times 2^{Act} \times S$ is the transition relation, AP is a set of atomic propositions and $L : S \to 2^{AP}$ is a state-labeling function.

An execution path of t is an infinite sequence of states and sets of actions $\rho = \langle s_0, \alpha_0, s_1\alpha_1, s_2, \alpha_2, s_3, ...\rangle$ such that $s_0 \in I$ and $(s_i, \alpha_i s_{i+1}) \in R$ for all $i \geq 0$.

This definition extends Kripke structures with a set of actions that can be part of a transition. In the context of a DSPL these actions represent reconfiguration operations required to adapt the system from a current configuration to a new desired configuration.

An L^2TS enables describing a DSPL's transition system considering reconfiguration operations. However, as mentioned earlier, completely specifying a DSPL transition system is unfeasible due to the large number of states and potential transitions. As in the case of temporal constraints, it would be more valuable to have a declarative language that enables describing in which cases reconfiguration operations are needed.

State/Event Linear Temporal Logic (SE-LTL) [23] is an extension of LTL to support expressing temporal properties over states and actions. SE-LTL is very similar to LTL, but allows for temporal logic state/event formulas over state atomic propositions and actions. Given $p \in AP$ and $a \in Act$, the syntax of an SE-LTL formula is defined by the following grammar:

$$\phi ::= p \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \to \phi \mid \phi \,\mathcal{U} \phi \mid \bigcirc \phi \mid \Box\phi \mid \diamond \phi$$

Semantics differ slightly as an SE-LTL property may also include constraints over actions. In these cases, an action holds at a given state in an execution path if it is part of the transition immediately after it. Thus, given an execution path $\rho = \langle s_0, \alpha_0, s_1, \alpha_1, s_2, ...\rangle$, action $a \in Act$ holds at state i iff $a \in \alpha_i$. Note that the transition relation $R \subseteq S \times 2^{Act} \times S$ is defined over the power set of actions and each transition can have multiple actions. The semantics of logical and temporal operators is the same as in LTL.

Using the SE-LTL we can express constraints from the motivating example given in Section II-A that involve operations and features. For instance, we could express the rule that changing the geographical region requires migrating the application as follows:

$$\Box(((US \wedge \bigcirc EU) \vee (EU \wedge \bigcirc US)) \to MigrateApp)$$

This constraint defines that whenever the location feature is changed from US to EU or vice-versa, the $MigrateApp$ operation should be executed. Therefore, by employing SE-LTL we can define temporal and propositional constraints over features and reconfiguration operations.

Considering temporal constraints and reconfiguration operations we can redefine the syntax and semantics of DSPLs based on a feature model.

Definition 7: Given a feature model $M = (\mathcal{F}, P)$ and a set of reconfiguration operations OP, we can define the corresponding DSPL as a doubly labeled transition system such that $DSPL(M, OP) = (S, I, Act, R, AP, L)$, where $I = S = P$, $Act = OP$, $R = S \times 2^{OP} \times S$, $AP = \mathcal{F}$ and $L(s) = s$.

This definition extends DSPLs based on a feature model with a set of reconfiguration operations. The obtained transition system allows transitioning between any feature model configuration by executing any combination of reconfiguration operations. As in the case of LTL formulas, SE-LTL can also be used to build a Büchi automaton [23] that can be combined with the transition system obtained from the feature model and reconfiguration operations to limit the allowed transitions.

Definition 8: Given a feature model M, a set of reconfiguration operations OP and a SE-LTL formula ϕ, we build a transition system $DSPL(M, OP, \phi) = DSPL(M, OP) \otimes \mathcal{A}_\phi$.

That is the synchronized product, as defined in [23], of the transition system obtained from the feature model and reconfiguration operations with the Büchi automaton that accepts the property defined by the formula ϕ.

D. Reconfiguration operation costs

Reconfiguration operations may result in downtime, performance loss, or additional costs during reconfiguration. In our motivating example, migrating a PostgreSQL database using `pgcopy` requires about 3 minutes of downtime per GB according to Heroku documentation. However, syncing a follower database may take hours, and adds the expense of a second database, but has minimal to no downtime.

To calculate the cost of applying a reconfiguration, we assume that cloud providers supply a set of cost functions $C = \{c_0, c_1, c_2, c_3, ...\}$, where each function $c_i \in C$ is specific to a type of cost (e.g., downtime, performance penalty, monetary cost) and takes as argument any reconfiguration operation $op \in OP$.

Since many cost functions require access to the current system configuration and load, we feel that providers are in the best position to provide such functions. However, many cost functions can be created, as we have done, by studying the provider's documentation. These serve as utility functions that can be used to optimize a higher-level fitness function that is written by the user. Furthermore, the units returned by a cost function (e.g., seconds, dollars, percentage) don't

need be standardized, nor do the same cost functions need to be applied across providers. Because users will invoke this functions explicitly, we only require that the user understand the function, the units returned by the function, and how to interpret them.

Introducing temporal constraints on DSPLs enables us to express detailed constraints on how variability can be applied to the system over its lifetime. Reconfiguration operations allow describing and analyzing the different ways of effecting a reconfiguration, as well as making explicit the constraints between features and operation. Finally, cost functions provide a means of understanding the side-effects of reconfigurations and for optimizing the user's extra-functional requirements, such as downtime or cost. Together these constructs support modeling adaptation constraints in cloud environments that cannot be captured with variability models.

IV. Reasoning on DSPL adaptation

The previous section presented how we define a DSPL as a transition system, and how temporal constraints can be combined with feature models to better capture a DSPL's adaptation behavior over time. In this section we discuss implementing a reasoner for identifying a reconfiguration plan given a reconfiguration request.

A. Reconfiguration query

Adaptations in a DSPL are usually triggered by requests to include or remove a set of features from the current configuration, or by changes in monitored context features. In both cases, we have a current system configuration and a high-level change request that specifies which features should be included or excluded. From these, we want to identify the set of possible reconfigurations that will move the system to a target state that conforms to the requested change. In addition, according to system preferences, we may also be interested in finding a reconfiguration that meets some criteria, such as having no downtime or minimizing costs.

Definition 9: A reconfiguration query $q = (A, E, \psi)$ is defined by a set A of features that should be part of the target configuration, a set E of features that should not be part of the target configuration, and a condition ψ over the reconfiguration transition. This condition can be defined over reconfiguration cost functions. Given $c \in C$ and $i \in \mathbb{N}$, a condition is defined over the following grammar:

$$\psi ::= \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \psi \rightarrow \psi \mid c\ relop\ i$$
$$relop ::= \ \leq \mid < \mid > \mid \geq \mid =$$

A condition ψ defines a predicate over transitions based on the estimated costs of executing their reconfiguration operations. Thus, given a condition $\psi = c\ relop\ i$, we have the predicate g_ψ over 2^{Act} defined as

$$g_q(\alpha) = \alpha \subseteq Act \wedge \left(\sum_{a \in \alpha} c(a)\right)\ relop\ i$$

For conditions that employ logical operators (i.e. $\wedge$, $\vee$, $\neg$, $\rightarrow$) we can obtain a corresponding predicate by applying these operators to predicates over costs accordingly.

Similarly, from feature sets A and E, we can also define a predicate over the set of states S of a transition system that recognizes candidate target configurations as

$$f_q(s) = s \in S \wedge A \subseteq L(s) \wedge (E \cap L(s) = \emptyset)$$

Given a transition system $T = (S, I, Act, R, AP, L)$, its current state s and reconfiguration query $q = (A, E, \psi)$, we can define the set of possible reconfigurations as

$$N_T(s, q) = \{(\alpha, s') \mid (s, \alpha, s') \in R \wedge g_q(\alpha) \wedge f_q(s')\}$$

A reconfiguration includes a target state and a set of reconfiguration actions for moving the system to the target states. The reconfigurations returned by function N_T are those for which the target states and reconfiguration actions comply with the predicates defined by the reconfiguration query.

Example

B. Optimization query

Besides finding a set of possible reconfigurations for a given query, one may be interested in identifying optimal reconfigurations. with minimum monetary costs or downtime.

Definition 10: An optimization query $o = (q, \phi)$ is defined by a reconfiguration query and an objective function ϕ. The objective function $\phi : OP \rightarrow \mathbb{N}$ can be defined as composition of cost functions $c \in C$.

Given a transition system $T = (S, I, Act, R, AP, L)$, its current state s and an optimization query $o = (q, \phi)$, we can define the set of optimal reconfigurations as

$$O_T(s, o) = \min_{(\alpha, s') \in N_T(s,q)} \left(\sum_{a \in \alpha} \phi(r)\right).$$

C. Symbolic representation

What's symbolic representation: From a system current configuration and reconfiguration query, we are interested in finding one or many possible target configurations. From the transition system of a DSPL, we can identify what are the possible reconfigurations by looking at the transitions outgoing the current system state and searching for the ones that agree with the reconfiguration query. However, as discussed earlier, explicitly building a DSPLs transition system can be unfeasible due to the large number of possible states and transitions.

One approach to deal with this problem is to build a symbolic representation of the transition system. The idea behind symbolic representation is to represents sets of system states and the transition relation as a propositional formula. Set operations can then be effected as logical operations between formulas and set membership by solving satisfiability. Thus, the set of states of a transition system can be defined by a propositional formula over the set of atomic propositions using substantially less space than the number of possible states.

A feature model can also be seen as a symbolic representation of the set of valid configurations for a system.

A model with n features can describe a system with up to 2^n possible configurations. In addition, a feature model can also be represented as a propositional formula [25], for which the satisfiable assignments represent the set of valid configurations.

Let $T = (S, I, Act, R, AP, L)$ be a transition system, the transition relation $R \subseteq S \times 2^{Act} \times S$ can be defined by a propositional formula over $AP \cup Act \cup AP'$. For representing a transition system as a propositional formula we need to define two sets of propositional variables, representing the set of atomic propositions AP at the source and target states. Here we have $AP' = \{p' \mid p \in AP\}$ as a copy of atomic propositions that represent the labeling in a target state. Besides this, we have to consider transition actions that may occur between any two states, represented by Act.

For each state in a transition system, the labeling function $L : S \to 2^{AP}$ defines exactly what are the atomic propositions that hold at a given state. For any state $s \in S$, we may define a propositional formula that uniquely identifies this state as

$\widetilde{L(s)} = (\bigwedge_{p \in L(s)} p) \wedge (\bigwedge_{p \in (AP \setminus L(s))} \neg p)$.

Similarly, every transition (s, α, t) in the transition relation $R \subseteq S \times 2^{Act} \times S$ has a set of transition actions $a \in \alpha$. For any set of transition actions, we may define a propositional formula that uniquely identifies it as

$\widetilde{L(\alpha)} = (\bigwedge_{a \in \alpha} a) \wedge (\bigwedge_{a \in (Act \setminus \alpha)} \neg a)$.

Relying on these functions that uniquely identify a set of actions and a state, we can define a transition relation by the following propositional formula

$\widetilde{R_t} = \bigvee_{(s,\alpha,t) \in R} \widetilde{L(s)} \wedge \widetilde{L(\alpha)} \wedge \widetilde{L(t)}'$.

Thus, from any state $s \in S$ the set of outgoing transitions can be obtained by the propositional formula

$\widetilde{Post(s)} = \widetilde{R_t} \wedge \widetilde{L(s)}$.

Based on this we can obtain a propositional formula that represents the set of transitions that can be obtained by a reconfiguration query. So, given a reconfiguration query $q = (A, E, \psi)$ and current state s, we can obtain the propositional formula

$\widetilde{Post(s, q)} = \widetilde{L(s)} \wedge \psi \wedge (\bigwedge_{p \in A} p') \wedge (\bigwedge_{p \in E} \neg p')$

that defines the set of transitions from state s that are according to query q. By solving satisfiability over the formula we can find the reconfiguration alternatives that support are according to query.

Here, we showed how we can represent a transition system symbolically by a propositional formula and use satisfiability solver to find a set of transitions that support a given query. However, we are not interested in building a symbolic representation from a complete transition system, but directly from a feature model extended with temporal constraints and reconfiguration operations.

As mentioned earlier, if we do not consider temporal constraints, a DSPL defined by a feature model can transition between any two valid configurations. So, given a feature model M, we have a propositional formula $\widetilde{M}$ that defines the set of allowed configurations $[\![M]\!]_{FM}$ and a set of operations OP, we can define the propositional formula for the transition relation as

$\widetilde{R_{FM}} = \widetilde{M} \wedge \widetilde{OP} \wedge \widetilde{M}'$.

An LTL formula can also be transformed into a propositional formula [26], [27], that symbolically defines an equivalent transition relation. The obtained propositional formula is defined over the atomic propositions included in the formula, which in the DSPL case can be features or operations. From an LTL formula ϕ, we obtain a propositional formula I_ϕ that defines the set of states in which ϕ holds and R_ϕ that defines the transition relation. Thus, from a feature model M and an LTL formula ϕ we can obtain the propositional formulas

$\widetilde{I_{M\phi}} = \widetilde{m} \wedge I_\phi$

that represents the set of valid configurations at the initial state, and

$\widetilde{R_{M\phi}} = \widetilde{m} \wedge \widetilde{m}' \wedge \widetilde{OP} \wedge R_\phi$

that represents the set of allowed transitions between system states. Similarly, the set of valid transitions from a given current state s is defined by the formula

$\widetilde{Post_{M\phi}}(s) = \widetilde{L(s)} \wedge \widetilde{R_{M\phi}}$.

Once we have a propositional formula that represents the set of valid transitions from a current state, we can use a SAT solver to find what are the possible transitions. So, given a reconfiguration query $q = (A, E, \psi)$ without any optimization goal or cost constraints ($\psi = true$), we can obtain a valid transitions by finding a satisfiable assignment to the formula

$\widetilde{Post_{M\phi}}(s, q) = \widetilde{L(s)} \wedge \widetilde{R_{M\phi}} \wedge (\bigwedge_{p \in A} p') \wedge (\bigwedge_{p \in E} \neg p')$.

If there are cost constraints ($\psi \neq true$), we can use support from Pseudo-Boolean SAT solvers [28] for encoding linear inequalities as boolean constraints. Similarly an optimization request can be encoded using Pseudo-Boolean SAT optimization [29].

Symbolic representation of transition systems enables to express the sets of states and the transition relation as a propositional boolean formula. From a given feature model and LTL formula, we can obtain a corresponding transition system as well as its symbolic representation. Given a reconfiguration query, we can use SAT solvers to reason over the system's symbolic representation and identify a valid transition from a current state.

V. Evaluation

To assess the use of temporal constraints and reconfiguration operations in DSPLs, we evaluate them in the context of a case study on the Heroku PaaS provider. Our goal is to validate the feasibility of the proposed approach in modeling configuration constraints of cloud providers, as well as the performance of evaluating reconfiguration requests.

A. Tool implementation

Before presenting the evaluation results, we give a brief overview of our implementation of the proposed approach.

To support specifying a DSPL, we designed a domain-specific language to define feature models with cross-tree

```
/* Feature Model */
Heroku {
 Application {
  Location [ US | EU | PrivateLocation [ Virginia | Oregon | Frankfurt | Tokyo | Sydney ] ]
   Process? {
    Buildpack [ Ruby | Node | Clojure | Python | Java | Gradle | Grails | Scala | Play | PHP | Go ]
    Kind [ Web | Worker ]
    Size [ Free | Hobby | Standard1X | Standard2X | PerformanceM | PerformanceL ]
   }
    Addons? {
    HerokuPostgres? [ HobbyDev | HobbyBasic | Standard0 | Standard2 | Premium1 | Premium2 ]
    ClearDBMySQL? [ Ignite | Punch | Drift | Scream ]
    /* ... */
   }
  }
 }
 /* ... */
}
/* Temporal constraints */
[]( (Punch -> X!Ignite) && (Drift -> X!(Punch || Ignite)) && (Scream -> X!(Drift || Punch || Ignite)) )
[]( Change(ClearDBMySQL) -> UpgradeMySQL )

[]( Change(Location) -> MigrateApp )

[]( (Change(HerokuPostgres) && (Switch(HobbyDev) || Switch(HobbyBasic))) -> PgCopy )
[]( Change(HerokuPostgres) -> (PgCopy || FollowerPgDb) )
```

Fig. 2. Sample extract from Heroku feature model

constraints on features, and temporal constraints on both reconfiguration operations and features. From the feature diagram and cross-tree constraints, we generate a propositional formula that correspond to a symbolic representation of the set of valid configurations for the DSPL. From temporal constraints, described as LTL formulas, we also obtain a symbolic representation of the DSPL's transition relation. These are then combined into propositional formulas that describe the DSPL's *transition system* as defined in Section IV. In both cases, we implemented well-known algorithms presented in previous work [25], [26], [27].

To perform perform a reconfiguration or optimization query, we convert the query into a propositional formula that represents the set of valid reconfigurations that comply with the query. We then find candidate reconfigurations by solving satisfiability over the combined formulas of the query and the DSPL's transition system.

As described in Section IV, the symbolic representation of transition systems enables to express the sets of states and the transition relation as a propositional boolean formula. We use the SAT4J[6] solver and the LogicNG[7] library for encoding Pseudo-Boolean constraints that involve cost functions. The implementation was largely written in Groovy with smalls parts in Java.

B. Case study

For our case study we designed a feature model containing the features and constraints described in Section II. which was augmented with data extracted from the Heroku website.

[6]http://www.sat4j.org/

[7]https://github.com/logic-ng

We obtained information on 161 *addon* cloud services from different categories, such as data storage, networking, security, management tools, etc. We automatically extracted, from each addon's webpage, the information on service plans, region availability and language support. From region availability information, we generated cross-tree constraints for 113 services that are not available in every region. Similarly, 66 constraints were generated from language support information. Additionally, for each addon with multiple service plans we generated a reconfiguration operation, and a corresponding temporal constraint, associating the operation to changing the addon plan.

The resulting Heroku feature model consists of 1036 features, 124 operations, 134 cross-tree constraints and 124 temporal constraints. We also defined two cost functions, *price* and *downtime*. that estimate, respectively, the monetary cost and the downtime of performing a reconfiguration operation. Fig. 2 shows an extract from this feature model, defined in the domain-specific language that was designed as part of our tool implementation. The extract includes a part of the feature model hierarchical structure as well as some temporal constraints between feature changes and reconfiguration operations.

C. Experimental setup

To evaluate reconfigurations in Heroku PaaS environments, we designed a set of 4 adaptation scenarios, defined in Table I, that represent example adaptations. For each scenario, we executed 5 different kinds of queries, including restrictions and optimizations on *price* and *downtime*. These experiments were run for 3 different utilization profiles that represent application metrics at the moment a reconfiguration is triggered. In our

TABLE I
EXPERIMENTAL SETUP

Adaptation scenarios	
1	Changes in database size triggers a request to a bigger database plan.
2	Request for a new feature that is not available in the current region.
3	Application requires changing the programming framework and database.
4	Application container needs to be scaled up.

Reconfiguration queries	
1	No cost constraints.
2	With constraints over downtime.
3	With constraints over downtime and price
4	Optimization based on downtime.
5	Optimization based on downtime and price.

Utilization profiles	
1	DBSize: 10GB, AppSize: 100 MB, AppStartUp: 15s
2	DBSize: 100GB, AppSize: 200 MB, AppStartUp: 30s
3	DBSize: 2TB, AppSize: 500 MB, AppStartUp: 60s

experiment, we considered application and database size, as well as application startup time. These metrics are used because they affect reconfiguration costs, but other information available from the executing environment could be used, in conjunction with new operation cost functions, as well.

For each combination of adaptation scenario, query and utilization profile we measured the time required to find one valid reconfiguration. Besides this, we also measured the time required to build a symbolic representation of a DSPL transition system from the Heroku feature model. We executed 12 times each combination, leading to a total of 720 individual query evaluations. All experiments were run on a late 2013 MacBook Pro Laptop with a 2 GHz Intel Core i7 processor and 8GB of memory.

D. Results

In Table II we present the minimum, maximum and average times of executing each of the measured processing operations. The first line shows the time required to build a symbolic representation of the transition system, which includes the time to translate the feature model diagram and LTL formulas into propositional formulas. The second line shows the time required to find a valid reconfiguration from a given query. This includes the time to translate the query into a propositional formula and solve satisfiability. The following lines shows results for executions grouped by the kind of query. As defined in Table I, we have reconfiguration queries without cost constraints (1), with cost constraints (2, 3) and optimization queries (4, 5).

The process step that required the most time was the construction of a symbolic representation for the transition system (average of 8777ms). However, this representation only needs to be built once and can then be used for executing multiple

TABLE II
RESULTS

	Execution time (ms)				
Process step	**Avg**	**StdDev**	**Min**	**Max**	**#Exec**
Build Trans System	8777.31	303.71	8262	10308	720
- Process FM	244.75	28.57	191	552	
- Process LTL	8533.57	291.81	8025	10023	
All Queries	183.34	50.20	118	389	720
- Build	83.05	50.69	27	227	
- Solve	100.29	37.13	5	200	
Q wo/ Constraints	140.97	12.93	118	198	144
- Build	32.73	4.56	27	48	
- Solve	108.24	11.58	90	153	
Q w/ Constraints	224.23	55.55	128	389	288
- Build	140.41	27.65	80	227	
- Solve	83.82	53.13	5	200	
Q w/ Optimization	163.63	13.26	136	230	288
- Build	50.85	7.11	38	77	
- Solve	112.77	10.21	94	166	

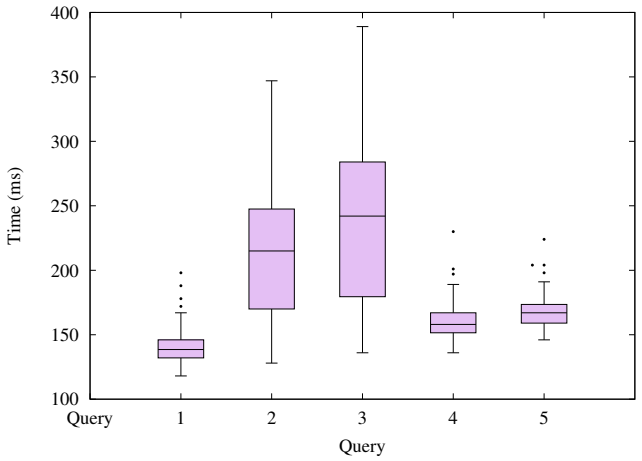

Fig. 3. Distribution times for executing each query 144 times (4 scenarios $\times$ 3 profiles $\times$ 12 runs)

reconfiguration queries. From analysis of the execution times for reconfiguration queries we can see that those including constraints on cost functions are the ones that required more time to be executed (average of 225.78ms). This is mainly due to the time required to encode Pseudo-Boolean constraints on cost functions as propositional formulas. For the other kinds of queries, most of time is spent on solving the boolean satisfiability problem that we encode.

Fig. 3, shows a distribution of execution time for different kinds of queries. From this chart, we can see that queries 2 and 3, that rely on cost constraints take substantially longer than other kinds of queries. Besides this, queries that combine different cost functions (3 and 5) take usually longer than with just one cost function (2 and 4).

E. Threats to validity

The feature model for Heroku used in our evaluation was built from information obtained through documentation and

structured data available on their website. This means the variability and constraints considered may not be exhaustive and there might be new constraints that are not supported by our approach. Also, the cost functions defined for our experiments return arbitrary values. More realistic cost functions should rely on better information retrieved from the cloud environment and the current configuration. Nevertheless, more precise cost functions (e.g., that return a better estimate), would not affect the performance or the modeling capacity of the approach since we only rely on the return values.

We evaluated our approach using one case study that applies to cloud computing. And although we have identified similar concerns in other systems where temporal constraints exist and are not modeled using unified or homogeneous mechanisms, we have not evaluated our approach in these cases. Hence, we are optimistic that our approach is applicable to other providers, to other scenarios, and to other domains, but this must be verified.

VI. Related work

Featured Transition Systems are a variant of transition systems designed to describe the combined behavior of an entire system family [30], [31]. Temporal logic is used for defining properties to be checked against systems that are part of a Software Product Line (SPL). However, their goal is to use model checking to verify temporal properties over a family of related systems, while ours is to define the adaptive behavior of a Dynamic Software Product Line (DSPL).

In many DSPL approaches, variation points are augmented with binding time information [12] to specify when features can or must be (de)activated, such as during compilation, configuration, start-up, runtime, etc. In [13], *Bürdek et al.* propose an approach for defining constraints over binding times, making it possible to indirectly define some temporal properties through constraints between features and binding times. However, in cloud environments, each cloud service may exhibit a different and independent lifecycle, and temporal constraints are not limited to binding time.

In [32], *reconfiguration actions* are used to update a system's architecture as a result of a dynamic reconfiguration. This concept is similar to reconfiguration operations used in our approach, but there is no support to evaluate the costs of actions nor to reason over multiple alternative reconfiguration paths.

In Genie [33], variability models are combined with transition diagrams that define how a system can adapt between its different variants. However, when the number of system variants is very large, fully defining its transition diagram can quickly become unfeasible. In our work, we employ temporal logic notation to define a system's allowed transitions, which enables to define the adaptive behavior in a declarative way.

In [34], authors propose to use temporal logic for specifying reconfigurable component-based systems. Similar to our work, temporal logic is used to declaratively define how a system can be reconfigured.

A few recent works propose the use of DSPLs to manage adaptive cloud systems. In [35] and [36], variability from applications and their running cloud environment are combined to build systems that can adapt their behavior and executing environment. In [37], authors propose an architecture for changing the structure and configuration options of feature models at runtime, allowing the dynamic evolution of the system's variability. Their work is also motivated by a cloud computing scenario.

Our approach shares with previous work the use of variability models and transition systems to support the modeling and management of adaptive systems. However, as cloud environments offer heterogeneous adaptation mechanisms there may also be complex constraints governing when and how a system can be adapted. Our work differs from previous work by combining variability models with temporal logic to support a declarative definition of adaptive systems with complex adaptation constraints.

VII. Conclusion

In this paper we propose the use of temporal constraints on variability models, which allows specifying a Dynamic Software Product Line's adaptive behavior. The need for temporal constraints arised from the complex reconfiguration constraints that we identified when managing the lifecycle of adaptive cloud systems. We integrated temporal properties and reconfiguration operations into the definition of DSPLs to support expressing these constraints and reasoning over them to find valid adaptations when a context change is identified.

Based on concepts from model checking, we implemented a tool for checking a reconfiguration query against the current system configuration. Our tool finds valid target configurations, as well as sets of reconfiguration operations required to update the cloud environment to the target configurations. We evaluated the approach with a use case for the Heroku PaaS cloud provider. The approach enabled us to define many existing constraints that were not supported by previous DSPL approaches and reason over them with low performance overhead. The prototype implementation and results can be found in the accompanying site[8].

We plan to investigate how these concepts can be applied to build multi-cloud adaptive environments, that is, cloud systems composed of services operating across different providers. In this case, reconfigurations may involve multiple providers and the system needs to coordinate multiple DSPLs with different variability and adaptation constraints. Another research direction is to use temporal properties to support variability evolution in cloud providers. As temporal properties define the allowed transitions in a system, they could be used to define valid reconfigurations for running systems when structural variability is updated.

References

[1] P. Mell and T. Grance, "The NIST definition of cloud computing," 2011.

[8] http://researchers.lille.inria.fr/sousa/seams17/

[2] C. Quinton, D. Romero, and L. Duchien, "SALOON: a platform for selecting and configuring cloud environments," *Software: Practice and Experience*, vol. 46, no. 1, pp. 55–78, 2016. [Online]. Available: http://dx.doi.org/10.1002/spe.2311

[3] G. Sousa, W. Rudametkin, and L. Duchien, "Automated setup of multi-cloud environments for microservices applications," in *9th IEEE International Conference on Cloud Computing*, San Francisco, United States, Jun. 2016.

[4] J. García-Galán, P. Trinidad, O. F. Rana, and A. Ruiz-Cortés, "Automated configuration support for infrastructure migration to the cloud," *Future Generation Computer Systems*, vol. 55, pp. 200 – 212, 2016.

[5] A. Ferreira Leite, V. Alves, G. Nunes Rodrigues, C. Tadonki, C. Eisenbeis, and A. Magalhaes Alves de Melo, "Automating resource selection and configuration in inter-clouds through a software product line method," in *8th IEEE International Conference on Cloud Computing*, New York, United States, Jun. 2015, pp. 726–733.

[6] B. Morin, O. Barais, J. M. Jezequel, F. Fleurey, and A. Solberg, "Models@run.time to support dynamic adaptation," *Computer*, vol. 42, no. 10, pp. 44–51, Oct 2009.

[7] A. Hubaux and P. Heymans, "On the evaluation and improvement of feature-based configuration techniques in software product lines," in *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, May 2009, pp. 367–370.

[8] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *COMPUTER*, vol. 41, no. 4, pp. 0093–95, 2008.

[9] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615 – 636, 2010.

[10] J. Mauro, M. Nieke, C. Seidl, and I. C. Yu, "Context aware reconfiguration in software product lines," in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, ser. VaMoS '16. New York, NY, USA: ACM, 2016, pp. 41–48. [Online]. Available: http://doi.acm.org/10.1145/2866614.2866620

[11] J. Lee and K. C. Kang, "A feature-oriented approach to developing dynamically reconfigurable products in product line engineering," in *10th International Software Product Line Conference (SPLC'06)*, 2006, pp. 10 pp.–140.

[12] R. Capilla and J. Bosch, *Binding Time and Evolution*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 57–73. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36583-6_4

[13] J. Bürdek, S. Lity, M. Lochau, M. Berens, U. Goltz, and A. Schürr, "Staged configuration of dynamic software product lines with complex binding time constraints," in *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, ser. VaMoS '14. New York, NY, USA: ACM, 2013, pp. 16:1–16:8. [Online]. Available: http://doi.acm.org/10.1145/2556624.2556627

[14] K. Mens, R. Capilla, N. Cardozo, and B. Dumas, "A taxonomy of context-aware software variability approaches," in *Companion Proceedings of the 15th International Conference on Modularity*, ser. MODULARITY Companion 2016. New York, NY, USA: ACM, 2016, pp. 119–124.

[15] R. Capilla, Ó. Ortiz, and M. Hinchey, "Context variability for context-aware systems," *Computer*, vol. 47, no. 2, pp. 85–87, Feb 2014.

[16] R. Capilla, M. Hinchey, and F. J. Díaz, "Collaborative context features for critical systems," in *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*, ser. VaMoS '15. New York, NY, USA: ACM, 2015, pp. 43:43–43:50. [Online]. Available: http://doi.acm.org/10.1145/2701319.2701322

[17] C. Cetina, P. Giner, J. Fons, and V. Pelechano, "Autonomic computing through reuse of variability models at runtime: The case of smart homes," *Computer*, vol. 42, no. 10, pp. 37–43, Oct 2009.

[18] K. Saller, M. Lochau, and I. Reimund, "Context-aware dspls: Model-based runtime adaptation for resource-constrained systems," in *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, ser. SPLC '13 Workshops. New York, NY, USA: ACM, 2013, pp. 106–113. [Online]. Available: http://doi.acm.org/10.1145/2499777.2500716

[19] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[20] K. Schneider, *Verification of Reactive Systems: Formal Methods and Algorithms*. SpringerVerlag, 2004.

[21] Z. Manna and A. Pnueli, "A hierarchy of temporal properties (invited paper, 1989)," in *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '90. New York, NY, USA: ACM, 1990, pp. 377–410. [Online]. Available: http://doi.acm.org/10.1145/93385.93442

[22] M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti, *An Action/State-Based Model-Checking Approach for the Analysis of Communication Protocols for Service-Oriented Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 133–148. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-79707-4_11

[23] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha, *State/Event-Based Software Model Checking*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 128–147. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24756-2_8

[24] C. Pecheur and F. Raimondi, *Symbolic Model Checking of Logics with Actions*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 113–128. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74128-2_8

[25] D. Batory, *Feature Models, Grammars, and Propositional Formulas*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 7–20. [Online]. Available: http://dx.doi.org/10.1007/11554844_3

[26] E. M. Clarke, O. Grumberg, and K. Hamaguchi, "Another look at ltl model checking," *Formal Methods in System Design*, vol. 10, no. 1, pp. 47–71, 1997. [Online]. Available: http://dx.doi.org/10.1023/A:1008615614281

[27] A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani, *Improving the Encoding of LTL Model Checking into SAT*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 196–207. [Online]. Available: http://dx.doi.org/10.1007/3-540-47813-2_14

[28] T. Philipp and P. Steinke, *PBLib – A Library for Encoding Pseudo-Boolean Constraints into CNF*. Cham: Springer International Publishing, 2015, pp. 9–16. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-24318-4_2

[29] E. Boros and P. L. Hammer, "Pseudo-boolean optimization," *Discrete applied mathematics*, vol. 123, no. 1, pp. 155–225, 2002.

[30] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: Efficient verification of temporal properties in software product lines," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 335–344. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806850

[31] A. Legay, P.-Y. Schobbens, P. Heymans, M. Cordy, J.-F. Raskin, and A. Classen, "Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking," *IEEE Transactions on Software Engineering*, vol. 39, no. undefined, pp. 1069–1089, 2013.

[32] C. Cetina, P. Giner, J. Fons, and V. Pelechano, "Using feature models for developing self-configuring smart homes," in *2009 Fifth International Conference on Autonomic and Autonomous Systems*, April 2009, pp. 179–188.

[33] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair, "Genie: Supporting the model driven development of reflective, component-based adaptive systems," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 811–814. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368207

[34] N. Aguirre and T. Maibaum, "A temporal logic approach to the specification of reconfigurable component-based systems," in *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, 2002, pp. 271–274.

[35] A. Almeida, E. Cavalcante, T. Batista, N. Cacho, F. Lopes, F. C. Delicato, and P. F. Pires, "Dynamic adaptation of cloud computing applications," in *The 25th International Conference on Software Engineering and Knowledge Engineering, Boston, MA, USA, June 27-29, 2013.*, 2013, pp. 67–72.

[36] A. Metzger, A. Bayer, D. Doyle, A. M. Sharifloo, K. Pohl, and F. Wessling, "Coordinated run-time adaptation of variability-intensive systems: An application in cloud computing," in *Proceedings of the 1st International Workshop on Variability and Complexity in Software Design*, ser. VACE '16. New York, NY, USA: ACM, 2016, pp. 5–11. [Online]. Available: http://doi.acm.org/10.1145/2897045.2897049

[37] L. Baresi and C. Quinton, "Dynamically evolving the structural variability of dynamic software product lines," in *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 57–63. [Online]. Available: http://dl.acm.org/citation.cfm?id=2821357.2821367