### ABSTRACT
Dynamic Software Product Lines (DSPL) have recently shown promising potentials as integrated engineering methodology for (self-)adaptive software systems. Based on the software-configuration principles of software product lines, DSPL additionally foster reconfiguration capabilities to continuously adapt software products to ever-changing environmental contexts. However, in most recent works concerned with finding near-optimal reconfiguration decisions, real-time aspects of reconfiguration processes are usually out of scope. In this paper, we present a model-based methodology for specifying and automatically analyzing real-time constraints of reconfiguration decisions in a feature-oriented and compositional way. Those real-time aware DSPL specifications are internally translated into timed automata, a well-founded formalism for real-time behaviors. This representation allows for formally reasoning about consistency and worst-case/best-case execution-time behaviors of sequences of reconfiguration decisions. The technique is implemented in a prototype tool and experimentally evaluated with respect to a set of case studies.

### CCS CONCEPTS
- Software and its engineering → Software product lines; Theory of computation → Timed and hybrid models; Verification by model checking; Computer systems organization → Real-time system specification.

### KEYWORDS
Dynamic Software Product Lines, Reconfiguration Decisions, Timed Automata

---

1 Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC '20, October 19–23, 2020, MONTREAL, QC, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7569-6/20/10...
https://doi.org/10.1145/3382025.3414945

---

1. What is the minimum/maximum time to switch on/off particular features when performing a reconfiguration?
2. What is the minimum/maximum time the system is allowed to use a particular set of features?
3. What is the minimum/maximum time a reconfigurable system is allowed to use a particular set of features?
4. What is the minimum/maximum time a reconfigurable system is allowed to deactivate a particular set of features?
Such real-time reconfiguration constraints impose entirely novel analysis problems during DSPL engineering, including real-time aware consistency checks and optimization goals for reconfiguration choices (e.g., finding the fastest possible reconfiguration to satisfy newly emerging contextual requirements).

**Problem Statement.** In order to tackle these issues, an integrated DSPL engineering methodology has to be enhanced, accordingly, in order to properly address the following problem statements.

- How to express real-time constraints for DSPL reconfiguration decisions in a concise and feature-oriented way?
- How to automatically analyze such models w.r.t. crucial consistency properties?

**Contributions.** In this paper, we propose a model-based approach for formally specifying and automatically analyzing real-time constraints of reconfiguration decisions for reconfigurable (self-)adaptive systems developed as DSPL. Our textual specification language is feature-oriented and enables developers to express different kinds of real-time requirements for DSPL reconfiguration processes in an expressive, yet graspable way. The resulting real-time aware DSPL specifications are internally translated into timed automata (TA) [2], a well-founded formalism for modeling and analyzing real-time behaviors. This representation facilitates automated reasoning about consistency properties expressed in a restricted form of TCTL (timed computational tree logics) [1] as well as real-time aware reconfiguration decisions. We implemented our technique in a prototype tool utilizing the Uppaal tool [7] for TA modeling and TCTL analysis and we experimentally evaluated the applicability of our approach with respect to a set of community case studies.

To summarize, this paper makes the following contributions.

- **Modeling Language.** We combine feature diagrams [15] with a feature-oriented textual language for concisely expressing real-time reconfiguration constraints for DSPL.
- **Automated Analysis.** We devise a compositional transformation schema for translating such DSPL specifications into a semantically equivalent timed automata model. This model enables precise analyzes of those specifications by employing the rich theory and mature tool support of timed automata.
- **Tool Support.** We provide tool support for our technique based on FeatureIDE [28] for feature modeling and the Uppaal tool for timed-automata analyses.
- **Experimental Evaluation.** We present experimental evaluation results gained from applying our tool to a collection of community case studies, manually enriched by real-time reconfiguration constraints.

**Verifiability.** To make our results reproducible, we provide the tool implementation and all experimental results as well as raw data on a supplementary web page².

## 2 BACKGROUND

In this section, we introduce an illustrative example used throughout this paper to motivate and describe our concepts.

---

²https://doi.org/10.5281/zenodo.4038317
features can be one of the following four possible types, as also illustrated in our example.

- Feature **Speed** is mandatory (black-filled circle). Hence, selection of Train implies selection of Speed.
- Features Link, Inside Tunnel and Train Nearby are optional (white-filled circle). These features can be, but do not have to be, selected in case their parent is selected.
- The or-group (black-filled arc) below feature Link denotes that at least one child from that group must be selected if the parent is selected.
- The alternative-group (white-filled arc) below feature Speed denotes that exactly one child from that group (i.e., Fast, Slow or Halt) must be selected if the parent is selected.

Further dependencies can be expressed by cross-tree edges such as require-edges (e.g., Ad Hoc only works for Train Nearby) and exclude-edges (i.e., LTE does not work Inside Tunnels).

**DSPL.** In case of statically configurable software product lines, a configuration (i.e., a feature selection for a desired software product) is chosen by a customer or user once and for all time, usually before going operational for the first time. In contrast, in case of a dynamic SPL such as the train controller, the (software) product must be prepared to be reconfigurable (i.e., to switch between configurations), even at run-time. Such reconfigurations might be caused by contextual changes of the running system (e.g., a train using LTE reaches a Tunnel). In the feature diagram in Fig. 2, we explicitly model contextual situations by means of contextual features being located in the right sub-tree of the feature tree [13]. In this way, we can use cross-tree constraints between those contextual features and proper system features located in the left sub-tree to specify all candidate (re-)configurations potentially satisfying a contextual change (e.g., switching from LTE to Ad Hoc or WAP). By further adding information about (non-)functional properties of the different possible configurations (e.g., latency of communication links), those models can be used for preplanning (presumably optimal) reconfiguration strategies for all kinds of possible contextual switches. To this end, we translate DSPL specifications into timed automata (TA) [2], denoting state-transition graphs enriched by real-time behavior (see Section 4). This enables us to automatically analyze consistency properties for DSPL specifications expressible as TCTL expression [1] using off-the-shelf model checkers for TA such as UPPAAL (see Section 4).

**Problem Statements.** The actual computational effort caused by performing such reconfigurations at run-time (e.g., in terms of permitted CPU time or communication latency) is often out of scope. However, this may cause serious or even fatal problems (e.g., missing a hard communication deadline) in case of critical application domains as considered in our example. This observation leads us to our problem statements, P1–P3, as follows.

- **P1:** Reconfiguration decisions may require different amounts of setup times, depending on the current configuration and the features involved.
- **P2:** The minimum/maximum time a system is allowed to reside in a configuration and/or to progress to another configuration may be restricted, depending on the current configuration and the features involved.
- **P3:** Different real-time constraints may influence or even conflict with each other, depending on the current configuration and the features involved.

We now propose an integrated approach tackling these problems.

## 3 Modeling Real-Time Reconfiguration Constraints

In this section, we formally introduce our constraint language for expressing real-time requirements for reconfiguration behavior of DSPL as described in the previous section. The proposed language is declarative and feature-oriented, by employing feature constraints in terms of propositional formulae over Boolean features variables to denote sets of configurations affected by constraint in a compact way. As an example, the feature constraint

$$LTE \land Fast \land \neg WAP$$

denotes three valid configurations \{Ad-Hoc, LTE, Fast, Train Nearby\}, \{LTE, Fast, Train Nearby\}, and \{LTE, Fast\} of the example in Figure 2. We will omit abstract features (represented by a lighter shade of blue in feature diagrams) in our set-based notation of configurations for convenience and define feature constraints as follows.

**Definition 3.1 (Feature Constraint).** Let \( F \) be a set of feature names and \( f \in F \). A feature constraint \( \lambda \) over \( F \) is inductively defined as

$$\lambda := f \mid \neg \lambda \mid \lambda \land \lambda \mid \lambda \lor \lambda \mid \text{true}.$$  

In particular, for every feature model \( fm \) over a set \( F \) of feature names, a semantically equivalent feature constraint \( \lambda_{fm} \) over a set of Boolean variables \( F \) can be constructed which exactly denotes the set of valid configurations of \( fm \) [5]. Hence, we will omit the graphical representation of a feature model and refer to \( fm \) in the following. Moreover, by \([\lambda] \subseteq 2^F\) we denote the set of all valid configurations, containing those subsets \( k \in [\lambda] \) of features of set \( F \) satisfying feature constraint \( \lambda \). In this regard, \([\lambda_{fm}]\) denotes the set of all valid configurations w.r.t. a feature model \( fm \).

Next, we use feature constraints to express real-time reconfiguration constraints. To this end, we combine two feature constraints \( \lambda_1 \) and \( \lambda_2 \) as \([\lambda_1 \rightarrow \lambda_2]\) to denote a reconfiguration constraint which holds for all reconfiguration steps leading from configuration \( k \in [\lambda_1] \) to a subsequent configuration \( k' \in [\lambda_2] \). As an example, \([Fast \rightarrow Halt]\omega\) denotes a reconfiguration constraint \( \omega \) (see below) which holds for all reconfiguration steps leading from an active configuration containing feature Fast to a subsequent configuration containing feature Halt. As reconfiguration constraint \( \omega \), we consider four basic types of real-time requirements which are each represented using a predefined keyword in our language.

- **Keyword setup** expresses that the execution of a reconfiguration step requires a certain amount of time (e.g., the reconfiguration from Fast to Slow takes at least 45 seconds).

- **Keyword reside** expresses the time a configuration is allowed to stay active (e.g., a configuration with Fast is only permitted for 30 minutes).

- **Keyword progress** expresses that a reconfiguration has to be executed within a certain period of time (e.g., if feature Link is not contained in a current configuration, then feature Halt must be activated after at most 30 seconds).
We define real-time requirements as follows.

**Definition 3.2 (Real-Time Requirement).** A real-time requirement \( \omega \) is defined by the following grammar:

\[
\omega = \text{setup} \geq n \mid \text{reside} \leq n \mid \text{progress} \leq n \mid \text{false}
\]

with \( n \in \mathbb{N} \), \( \geq \in \{ \geq, >, \geq \} \) and \( \leq \in \{ <, \leq, \leq \} \).

We restrict real-time requirements to contain bounds \( n \in \mathbb{N} \), as real-valued bounds would obstruct fundamental decidability properties of timed automata (see below). By combining feature constraints (e.g., \([\text{Fast} \rightarrow \text{Slow}]\)) and real-time requirements (e.g., \( \text{setup} \geq 45 \)), we are now able to address problem statements \( P1 \) and \( P2 \) by means of real-time reconfiguration constraints. For instance, a minimum time required for decelerating a train is expressed as:

\[
[\text{Fast} \rightarrow \text{Slow}] \text{ setup} \geq 45.
\]

This constraint requires every reconfiguration step from a current configuration \( k \in \{\text{Fast}\} \) to a subsequent configuration \( k' \in \{\text{Slow}\} \) to take at least 45 time units (e.g., seconds).

We combine the previously described concepts to define real-time reconfiguration constraints as follows.

**Definition 3.3 (Real-Time Reconfiguration Constraint).** Let \( \lambda_{\text{fm}} \) be an FM over feature set \( F \). A real-time reconfiguration constraint \( \psi \) is defined by the following grammar:

\[
\psi = [\lambda \rightarrow \lambda'] \omega
\]

where \( \psi \) is a real-time requirement and \( \lambda, \lambda' \) are feature constraints over \( F \) such that \( \llbracket \lambda \rrbracket \cap \llbracket \lambda' \rrbracket = \emptyset \) holds.

We think that it is reasonable to require \( \llbracket \lambda \rrbracket \) and \( \llbracket \lambda' \rrbracket \) to be disjoint as, otherwise, evaluation semantics of constraints regarding configurations \( k \in \llbracket \lambda \rrbracket \cap \llbracket \lambda' \rrbracket \) may be undefined. As an example, consider the constraint

\[
\psi = [\text{InsideTunnel} \rightarrow \text{WAP}] \text{ setup} \geq 15.
\]

Here, a reconfiguration from \( k = [\text{InsideTunnel}, \text{WAP}, \text{Slow}] \) to \( k' = [\text{InsideTunnel}, \text{WAP}, \text{Fast}] \) would be affected by the constraint \( \psi \) as \( k \in \llbracket \text{InsideTunnel} \rrbracket \) and \( k' \in \llbracket \text{WAP} \rrbracket \). This seems counterintuitive as the features \( \text{InsideTunnel} \) and \( \text{WAP} \) are actually not reconfigured. Nonetheless, \( \psi \) would lead to a different behavior of this reconfiguration. Therefore, we explicitly exclude constraints with overlapping sets of source and target configurations.

We call the proposed language real-time reconfiguration constraint language (RRL) and assume a DSPL specification to consist of a feature model and a set of RRL constraints in the following. In the next section, we tackle \( P3 \) by proposing a technique for automatically analyzing consistency of such a DSPL specification.

**4 ANALYZING REAL-TIME RECONFIGURATION CONSTRAINTS**

Our concept for tackling \( P3 \) is based on reconfiguration automata (RA) [23]. The set of states of RA correspond to the set of valid configurations of the DSPL under consideration and the set of edges between states represent permitted reconfiguration steps. In this basic DSPL model, reconfiguration steps require zero time. In order to enrich RA by real-time reconfiguration behaviors, we employ the theory of timed automata (TA) [2].

**Basic Notions of TA.** Let us again consider our running example. For the sake of readability, we consider as our FM only the alternative-group of feature \( \text{Speed} \) in Figure 2. Thus, we have

\[
\llbracket \lambda_{\text{fm}} \rrbracket = \{[\text{Fast}], \{\text{Slow}\}, \{\text{Halt}\}\}
\]

comprising three valid configurations. A TA modeling the real-time reconfiguration behavior of this simplified example is depicted in Figure 3, referred to \( \text{TA}_{\text{FM}} \) in the following. Like in reconfiguration automata, each valid configuration \( k \in \llbracket \lambda_{\text{fm}} \rrbracket \) is, again, represented by a state which is, however, called location in TA. Consequently, we will use the terms configuration and location interchangeably in the context of \( \text{TA}_{\text{FM}} \). Furthermore, as initial location, we assume \( \text{[Fast]} \) (depicted by the arrow below the location). Each possible reconfiguration is represented by a corresponding edge (called switch in TA) being labeled with an action \( \sigma \in \Sigma \). Here, we have \( \Sigma = \{\text{fast}, \text{slow}, \text{halt}\} \), simply denoting the target location (i.e., target configuration) of the reconfiguration related to the switch. This sample \( \text{TA}_{\text{FM}} \) actually permits all possible reconfigurations between any pair of valid configurations.

By using TA as reconfiguration model, we can express real-time constraints for single reconfiguration steps as well as entire reconfiguration sequences. To this end, TA models are equipped with a set of clocks \( C \), acting as continuously and synchronously increasing, yet independently resetable numerical variables over clock domain \( T_C \) (e.g., \( T_C = \mathbb{R}_+ \)). A clock \( c \in C \) can be used to measure the time elapsed since the last reset of that clock. Thereupon, clock constraints restrict time behavior in sequences of action occurrences to denote valid timed runs. For example, consider \( \text{TA}_\varphi \) in Figure 4 in which clock \( x \) is used in two ways:

- The clock reset \( x \rightarrow 0 \) performed by the switch from location \( \text{Fast} \) to \( \text{AfterReset} \) sets the value of \( x \) back to 0.
- The guard \( \{x > 45\} \) attached to the right switch from location \( \text{AfterReset} \) to \( \text{~Fast} \) denotes that this switch can only be taken if the value of clock \( x \) is greater than 45.

In addition to switch guards, clock constraints can be also attached to locations to denote invariants which must hold while residing in that location during a timed run. For instance, the invariant \( x < t_{\text{reside}} \) of the upper right location \( \lambda' \) in Figure 7 only allows runs to reside in this location while the value of clock \( x \) is less than \( t_{\text{reside}} \). Further note that \( \text{TA}_\varphi \) also contains a special action symbol \( \tau \) denoting internal actions (see below), where we write \( \Sigma_{\tau} = \Sigma \cup \{\tau\} \). To sum up, we formally define a TA according to [2].

**Definition 4.1 (Timed Automaton).** A Timed Automaton (TA) over a set of clocks \( C \) is a tuple (\( L, \Sigma, \Sigma_{\tau}, I, \rightarrow \)), where

- \( L \) is a finite set of locations,
- \( \varnothing \in L \) is the initial location,
- \( \Sigma \) is a finite set of actions,
- \( I : L \rightarrow B(C) \) assigns invariants to locations, and
- \( \rightarrow \subseteq L \times B(C) \times \Sigma_{\tau} \times \mathbb{Z}^C \times L \) is a finite set of switches, where the set \( B(C) \) of clock constraints \( \varphi \) is inductively defined as

\[
\varphi := c \sim n \mid c \sim c' \sim n \mid \varphi \land \varphi \mid \text{true},
\]

with \( c, c' \in C, n \in T_C \), and \( \sim \in \{<, \leq, \geq, >\} \).
Furthermore, we use the notation \( t^{a,R} t' \) to denote that \((t, a, R, t') \in \rightarrow \) holds. Next, we describe a translation of a DSPL specification as introduced in Sect. 3 into an equivalent TA model.

**Translating RRCL constraints into TA.** Similar to reconfiguration automata, we represent the set of configurations of a DSPL as locations in \( TA_{FM} \). We refer to the resulting \( TA_{FM} \) as core DSPL reconfiguration model, initially permitting all possible \((re-)configurations without any real-time restrictions. Thereupon, we refine the core model to only allow the reconfiguration behavior specified by the set of RRCL constraints. At first sight, a straight-forward solution appears to be to simply enrich the core model by further clock constraints. However, this approach turns out to become over-complicated and non-compositional in case of multiple, potentially interdependent RRCL constraints. Instead, we construct for each RRCL constraint \( \psi \) a separate TA model \( TA_{\psi} \) running in parallel to the core model and interacting via synchronization actions in order to restrict the reconfiguration behavior as requested by \( \psi \). To this end, we employ parallel composition \([30]\) of TA models.

**Parallel Composition of TA.** Under parallel composition, a pair of TA synchronizes their behavior via shared actions. For example, assume \( TA_{FM} \) and \( TA_{\psi} \) both reside in their initial location (i.e., \([Fast]\) and \( Fast \)). If \( TA_{FM} \) executes the switch labeled with \( halt \), then \( TA_{\psi} \) must also fire its switch labeled with the same action such that both TA synchronously enter the target locations, \([Halt] \) and \(-Fast \), respectively. In contrast, executions of switches labeled with \( \tau \) (e.g., from \( Fast \) to \( AfterReset \) in \( TA_{\psi} \)) do not require any synchronization thus enabling one TA to proceed to another location independently of the TA running in parallel. Based on the notion of composition, we can model each RRCL constraint in a separate TA synchronizing with the core model \( TA_{FM} \), which facilitates a compositional construction method. We next describe construction schemas for the different kinds of RRCL constraint in more detail.

**Construction Schema for Setup-Constraints.** Let us consider the setup-constraint

\[
\psi = [Fast \rightarrow Slow] \text{ setup } \geq 45
\]

supposed to affect the switch \([Fast] \xrightarrow{slow} [Slow] \) in the core model \( TA_{FM} \). This reconfiguration step should take at least 45 seconds. To enforce \( \psi \), we construct \( TA_{\psi} \) as shown in Figure 4 with location \( Fast \) being the initial location. As a consequence, if \( TA_{FM} \) and \( TA_{\psi} \) reside in their initial locations, a reconfiguration to \([Slow]\) is not possible. The disabling of the respective switch in \( TA_{FM} \) is achieved by \( TA_{\psi} \) not providing action \( slow \) in its initial location \( Fast \). In contrast, after executing the \( \tau \)-labeled switch to \( AfterReset \), \( TA_{\psi} \) is able to permit \( TA_{FM} \) to execute a \( slow \)-labeled switch as soon as the guard \([x > 45] \) is satisfied. In this way, \( TA_{\psi} \) influences the reconfiguration behavior of \( TA_{FM} \) in a way that satisfies constraint \( \psi \).

This construction schema can be generalized to all kinds of setup-constraints of the form:

\[
\psi_i = [\lambda \rightarrow \lambda'] \text{ setup } > t_{setup}.
\]

Index \( i \) is attached to \( \psi_i \) to identify the corresponding RRCL constraint. Based on \( \psi_i \), we explain how \( TA_{FM} \) is constructed in Figure 5. The configuration space (i.e., set of all valid configurations \([\lambda_{FM}] \)) is partitioned into three subsets by \( \psi_i \): The first subset contains configurations that either satisfy \( \lambda \), \( \lambda' \) or neither of both (i.e., \( k \in [[\neg \lambda \land \neg \lambda']] \)). A dashed rectangle in Figure 5 does not correspond to one single location of \( TA_{FM} \) but rather to a set of locations (i.e., dashed rectangles depict abstract locations). The same applies to arrows: an arrow from \([\lambda] \) to \([\lambda'] \) denotes that \( TA_{FM} \) contains a switch between all locations of the respective sets:

\[
\forall k \in [[\lambda]], k' \in [[\lambda']] : \exists k' \rightarrow k'
\]

Each switch is annotated with action (\( \lambda_i, \lambda_i' \) or \( not_i \)) for synchronization with \( TA_{\psi} \), see Figure 6. This TA looks similar to that in Figure 4 except for the names of locations and actions which are supposed to reflect the feature constraints \( \lambda \) and \( \lambda' \) of \( \psi_i \). For instance, in both \( TA_{FM} \) and \( TA_{\psi} \) every switch leading to a location associated with \( \lambda \) is annotated with action \( \lambda_i \). The names of locations in \( TA_{\psi} \) depict feature constraints corresponding to the current configuration (i.e., location) of \( TA_{FM} \). For instance, assume \( TA_{\psi} \) to reside in location \( \neg \lambda \). Our construction schema ensures that \( TA_{FM} \) is either in \([\neg \lambda \land \neg \lambda'] \) or \([\lambda'] \) as all these configurations satisfy \( \neg \lambda \). A reconfiguration from one of their locations to \([\lambda] \) executes action \( \lambda_i \). Thus, \( TA_{\psi} \) executes the switch from \( \neg \lambda \) to \( \lambda \) to synchronize with \( TA_{FM} \) on \( \lambda_i \) thus restricting the behavior of \( TA_{FM} \). Next, we present our construction schema for reside-constraints.

![Figure 3: TA_{FM} modeling the configuration space of an FM](image1)

![Figure 4: TA_{\psi} for the constraint \([Fast \rightarrow Slow]\) setup \( \geq 45 \)](image2)

![Figure 5: TA_{FM} modeling the configuration space of an FM](image3)
After not \( \lambda \) via action \( k \) has been entered via a preceding configuration consider the location \( \lambda \) of the form: \( \left[ \right] \). In this case, the DSPL must reconfigure within 30 seconds to a configuration containing \( \text{Halt} \) (e.g., an emergency stop of the train if no communication link is available).

We now consider all kinds of \( progress \)-constraints of the form:

\[
\psi_i = [\lambda \rightarrow \lambda'] progress < t_{progress}
\]

In general, a \( progress \)-constraint enforces the DSPL to leave configurations from set \( \left[ \right] \) to some configuration satisfying \( \lambda' \) within \( t_{progress} \) time units. For this, the DSPL is not allowed to leave \( \left[ \right] \) by switching to a configuration \( k \in \left[ \right] \) to prevent the DSPL from circumventing the intended reconfiguration behavior (e.g., an emergency stop could be circumvented by switching Link continuously on and off).

Next, we describe how to enforce those constraints in TA. Core model \( TA_{FM} \) (Figure 5) is enriched by synchronization actions in the same way as for \( setup \)- and \( reside \)-constraints. In \( TA_{\Psi} \) (see Figure 8), every location is related to exactly one abstract location in Figure 5. Thus, every reconfiguration targeting a configuration \( k \in \left[ \right] \) synchronizes with one of the switches annotated with \( \lambda \) of \( TA_{\Psi} \). Invariant \( x < t_{progress} \) of location \( \lambda \) then ensures that this location has to be left with \( t_{progress} \) time units. Furthermore, a reconfiguration to \( \left[ \right] \) is guaranteed as location \( \lambda \) has only one outgoing switch.

So far, we described construct schemas \( TA_{\Psi} \) for every type of constraint \( \psi \) individually. Concerning the whole set \( \Psi \) of RRCL constraints of a DSPL specification, our construction schemas permit a compositional approach such that the construction of each individual \( TA_{\Psi} \) for any \( \psi \in \Psi \) is independent of the other constraints in \( \Psi \). However, this does not hold for the modifications to be done in \( TA_{FM} \) which will be explained in the following.

**Handling Sets of RRCL Constraints.** In case of multiple constraints potentially affecting overlapping sets of configurations, TA construction may require switches in \( TA_{FM} \) being labeled with multiple synchronization actions. For instance, the two RRCL constraints

- \( \psi_1 = [\text{Fast} \rightarrow \text{Slow}] setup \geq 45 \), and
- \( \psi_2 = [\text{Slow} \rightarrow \text{Fast}] reside \leq 1800 \)

both introduce different actions at the switch from \{Fast\} to \{Slow\} to adapt their location to a reconfiguration from \{Fast\} to \{Slow\} (depicted in Figure 9). Unfortunately, TA do only allow one action-label per switch. To solve this issue, we introduce for \( n \) different actions affecting the same switch in \( TA_{FM} \), \( n - 1 \) intermediate locations as depicted in Figure 10. Here, \( m \) is a fresh clock which is reset by every incoming switch of each intermediate location. The passage of time is suspended while residing in these intermediate locations by attaching the invariant \( m \leq 0 \). Hence, all affected actions (e.g.,...
Figure 9: Switch with multiple actions (disallowed)

Figure 10: Representation of a switch with multiple actions

\( \lambda'_1 \) and \( \lambda_2 \) are executed in some order, but with zero delay in between. This schema emulates multiple TA\(_\psi\) to synchronize on the same reconfiguration in TA\(_{FM}\). As a result, every TA\(_\psi\) is able to either permit or prohibit the corresponding reconfiguration independent from all other TA\(_\psi\), thus potentially leading to complicated interactions between RRCL constraints.

To summarize, we presented in this section a construction schema of a TA model reflecting the real-time reconfiguration semantics of a DSPL specification consisting of a feature model and a set of RRCL constraints \( \Psi \). Next, we outline the potential semantic interplay of constraints which may yield to inconsistencies in the model. For example, consider the three constraints

- \( \psi_1 = [\text{Fast} \rightarrow \text{Slow}] \) setup \( \geq 45 \),
- \( \psi_2 = [\text{Slow} \rightarrow \text{Halt}] \) setup \( \geq 30 \), and
- \( \psi_3 = [-\text{Link} \rightarrow \text{Halt}] \) progress \( \leq 60 \).

Furthermore, assume the current configuration to be \{Link, LTE, Fast\}. If LTE becomes unavailable, both LTE and Link are deselected. In this case, \( \psi_3 \) requires the system to select Halt within 60 seconds (i.e., to perform an emergency stop if no communication link is available). However, the combination of \( \psi_1 \) and \( \psi_2 \) makes it impossible to satisfy \( \psi_3 \) as both setup times sum up to 75 seconds for a reconfiguration from [Fast] to [Halt]. Thus, the interplay between all three constraints results in an inconsistent DSPL specification. To detect inconsistencies, we describe how the TA representation allows us to verify correctness properties of DSPL specifications in an automated way. To this end, we first recapitulate operational semantics of TA based on Timed Labeled Transition Systems (TLTS) [14].

Operational Semantics of TA. For instance, the TLTS corresponding to the TA depicted in Figure 10 is partially shown in Figure 11. A TLTS consists of \( \text{states} \) being pairs of a TA location and a valuation for all clocks \( v : C \rightarrow \mathbb{T}_C \) (where we write \( m = v(m) \), for convenience, to denote that clock \( m \) has value \( v(m) \)). For example, the initial state of the TLTS consists of location \{Fast\} and clock valuation \( m = 0 \). Furthermore, a TLTS has two types of transitions:

- **Delay transitions** represent passage of time while staying in a TA location. For instance, a transition from \( \langle \text{Fast}, m = 0 \rangle \) to \( \langle \text{Fast}, m = 1 \rangle \) corresponds to the passage of one time unit in location [Fast].
- **Action transitions** represent execution of actions when taking a switch from one location to another location without any time elapsing. For instance, the transition from \( \langle \text{Fast}, m = 1 \rangle \) to \( \langle t', m = 0 \rangle \) corresponds to the execution of the switch to \( t' \) which resets clock \( m \).

To sum up, TLTS semantics of a TA is defined as follows [2].

**Figure 11: Excerpt of the TLTS for the TA in Figure 10**

**Definition 4.2 (TLTS).** The Timed Labeled Transition System (TLTS) of a TA \((L, \ell_0, \Sigma, I, \rightarrow)\) is a tuple \((S, s_0, \Sigma_r, \rightarrow)\), where

- \( S = L \times \mathbb{U} \) is a set of states with initial state \( s_0 = (\ell_0, [C \mapsto 0]|u_0) \in S \),
- \( \Sigma_r = \Sigma_r \cup \Delta \) is a set of labels with \( \Delta = \mathbb{T}_C \), \( \Sigma_r \cap \Delta = \emptyset \), and
- \( \rightarrow \subseteq S \times \Sigma_r \times S \) is a set of transitions being the least relation satisfying the rules:

\[
\begin{align*}
&\langle (t, u) \rangle \overset{d}{\rightarrow} \langle (t, u + d) \rangle \quad \text{if} \quad u \in I(t) \quad \text{and} \quad (u + d) \in I(t) \quad \text{for} \quad d \in \mathbb{T}_C, \\
&\langle (t, u) \rangle \overset{\sigma}{\rightarrow} \langle (t', u') \rangle \quad \text{if} \quad t \in I(t'), \quad u \in g, \quad u' \in I'(t'), \quad \sigma \in \Sigma_r.
\end{align*}
\]

A TLTS of a TA is, in general, not finite but allows us to illustrate operational semantics of TA in an intuitive way. TA analysis tools like the model checker Uppaal [7], however, internally utilize a finite symbolic representation of TA semantics. In particular, Uppaal is able to check properties defined by a query language based on Timed Computation Tree Logic (TCTL) [1]. A query in Uppaal is composed of path formulae and state formulae [6].

A TLTS of a TA is, in general, not finite but allows us to illustrate operational semantics of TA in an intuitive way. TA analysis tools like the model checker Uppaal [7], however, internally utilize a finite symbolic representation of TA semantics. In particular, Uppaal is able to check properties defined by a query language based on Timed Computation Tree Logic (TCTL) [1]. A query in Uppaal is composed of path formulae and state formulae [6]. The latter ones are used to define predicates over the states of a TLTS. Consequently, a state formula is evaluated over a state of the TLTS yielding either true or false. For instance, consider TA\(_\psi\) depicted in Figure 4. Here, the state formula

\[ \phi = \text{TA}_\Psi.Fast \]

is only satisfied in states of the TLTS of TA\(_\psi\) being related to location Fast (e.g., \( \langle \text{Fast}, x=3 \rangle \)). Furthermore, state formulae may also contain operators of propositional logic (e.g., and, or, imply, not). Besides state formulae, path formulae are used to define which paths of the TLTS are relevant for the query. Moreover, path formulae are always related to a property \( \phi \) (e.g., a state formula).

We next describe crucial semantic properties of DSPL specifications which we are now able to specify and check by applying TCTL formulae to the TLTS of the constructed TA representation.

**Deadlock Freedom.** After a sequence of reconfiguration steps, it may happen that eventually a configuration is reached which cannot be left any more (e.g., due to contradicting RRCL constraints, no more reconfiguration is possible). Such a deadlock can be detected by Uppaal with a query of the form:

\[ \text{E} \leftarrow \text{deadlock and (TA\_FM.Fast or TA\_FM.Slow or TA\_FM.Halt)} \]

Here, the path formula \text{E} \leftarrow \text{deadlock and (TA\_FM.Fast or TA\_FM.Slow or TA\_FM.Halt)} instructs Uppaal to find a state on an arbitrary path in the TLTS satisfying the corresponding state formula (i.e., deadlock and...). Thus, the query only yields a negative result if it is impossible to satisfy the state formula. Keyword deadlock is a built-in construct of Uppaal syntax for identifying deadlock states. Furthermore, the disjunction instructs Uppaal to only take into account proper locations in TA\(_{FM}\) and to ignore intermediate locations.
Reachability. A location being unreachable from the initial location in $TA_{FM}$ is another indicator for an undesirable inconsistency in DSPL specifications. To detect those locations, the following query can be used in Uppaal:

$$E<> TA. FM. Halt$$

If the query succeeds, then there exists at least one path ($E<>$) eventually reaching the given location (e.g., $TA. FM. Halt$).

Liveness. Finally, liveness of reconfiguration behavior is another crucial property of DSPL specifications. Unfortunately, Uppaal only supports a limited subset of TCTL forbidding nested path formulae as required for more elaborated liveness properties. Hence, we currently limit our considerations to check whether each pair of locations is mutually reachable from each other.

In the following section, we present experimental results gained from applying our TA construction and verification methodology for DSPL analysis to a collection of subject systems.

5 TOOL IMPLEMENTATION & EVALUATION

In this section, we present a tool implementation of our approach as well as results of an experimental evaluation obtained from applying our tool to a collection of subject systems. The goal of the experiments is (1) to show applicability of the proposed approach as well as (2) to investigate the computational effort required for the different steps of the methodology.

Tool Support. We implemented the methodology in the tool RRCL-Analyazer. Our tool takes a DSPL specification, consisting of an FM in the file format of FeatureIDE and a set of RRCL constraints in a text-based representation, as inputs (see Section 3). Our tool then constructs a set of TA as described in Sect. 4. The generated output is written into an XML file serving as input for the TA tool Uppaal which is executed with respective analysis queries for consistency checking. The execution times required for the overall procedure can be measured and logged, accordingly, to support experimental evaluation. We provide the tool, case studies and experimental results on a supplementary web page.

Research Questions.

- **RQ1.** What is the computational effort for generating a TA representation from a DSPL specification as compared to the subsequent consistency checks?
- **RQ2.** How does the computational effort for consistency checks depend on the size of the FM?
- **RQ3.** How does the computational effort for consistency checks depend on the number of RRCL constraints?

Methods and Experimental Design. Due to the novelty of the modeling approach, no DSPL case studies with RRCL constraints exist so far. Hence, we used a collection of existing FM models as well as synthetically generated FM and augmented them with RRCL constraints, synthetically generated by a self-written RRCL generator. This allows us to control the sizes of FM for answering RQ2 as well as the number of RRCL constraints for answering RQ3. For answering RQ1, we measured CPU time required for TA generation from the subject systems considered for RQ3. Our RRCL generator samples feature constraints (i.e., $\lambda$ and $\lambda'$), constraint type (e.g., setup) and corresponding time intervals from a uniform distribution. For creating feature constraints, we use a publicly available SAT generator\(^3\). The resulting DSPL specifications (i.e., FM and RRCL constraints) serve as input for our tool which automatically performs the TA construction and calls Uppaal for consistency checks. To keep the overall experimental setting graspable, we limit our considerations to reachability checks. However, the results for checking deadlock freedom and liveness are available on the supplementary web page.

Subject Systems. We applied RRCL-Analyazer to our running example as well as two further case studies from literature. The first case study, the database system BerkeleyDB [25], has 256 possible valid configurations, as the FM consists of 8 independent features. The second case study, called Hebok [26], represents a cloud scenario, namely a platform-as-a-service environment with 25 features and 720 possible valid configurations. We further enlarge the set of subject systems by 75 synthetically generated models using BeTTY [24], each having at most 256 valid configurations. BeTTy produced 75 FM with 2 up to 246 valid configurations. We additionally augment each FM with different numbers of RRCL constraints (i.e., $|\Psi| \in \{0, 3, 6, 9\}$). As a result, we gained 308 subject systems by means of DSPL specifications.

Data Collection. For each subject system, we run our tool to first generate the corresponding TA representation. We then apply the checks presented in Section 4 by calling Uppaal. For both steps, we measure the required CPU times. For the first step, this time includes parsing of the DSPL specification up to the output of the XML file. For the seconds step, we measured the time between the startup and termination of Uppaal. We repeated each experimental run three times and took the average value.

Measurement Setup. We executed all experiments on an Intel Core i7-6700k machine with 4x4GHz, 16GB RAM and Linux Mint 19.2. Our tool is implemented in Java (Azul OpenJDK 1.8.0_202). For the consistency checks, we call Uppaal version 4.1.19.

5.1 Results and Discussion

We now present experimental results grouped by RQ.

**RQ1.** For all subject systems, we obtained values between 866 ms and 1,218 ms on average for TA generation.

**RQ2.** Measurements for RQ2 regarding reachability analysis are shown in Figure 12. Here, the subject systems (i.e., FMs) are sorted according to the number of valid configurations. The y-axis depicts the measured CPU time in seconds and the black circles represent the average value of three measurements. Note that the reachability check yields positive results for every subject system of RQ2 due to the absence of reconfiguration constraints.

**RQ3.** Measurements for RQ3 regarding reachability analysis are shown similar to Figure 12 in Figure 13. Additionally, we distinguish the results for different $|\Psi| \in \{0, 3, 6, 9\}$ by different symbols. Note that, in contrast to RQ2, reachability checks now yield both positive and negative results: 83 out of 308 subject systems were reported.

---

\(^3\)https://github.com/RalfRothenberger/Power-Law-Random-SAT-Generator Accessed: 2020-03-18
inconsistent (i.e., not every valid configuration is reachable due to contradicting RRCL constraints).

Next, we discuss the results w.r.t. to each RQ separately.

**RQ1.** As a first observation, especially in case of larger FMs (in terms of $|\Psi|$), the effort for TA generation is negligible as compared to the analyses runs of Uppaal (e.g., 1.2 seconds versus 35 seconds for $|\Psi| = 246$). Thus, applicability of our approach mostly depends on scalability of the checks in Uppaal.

**RQ2.** The measurements for reachability analysis for all subject systems with $|\Psi| = 0$ are depicted in Figure 12. As a baseline, we also considered DSPL specifications without any RRCL constraints ($|\Psi| = 0$). Except for one outlier at $|\Psi| = 30$, the values seem to adhere an exponential curve. This can be explained by the fact that the number of configurations corresponds to the number of TA locations of $\Psi_{FM}$ which is (besides the number of clocks) the major factor influencing the complexity of TA model checking (e.g., reachability checking for TA is PSPACE-hard [2]).

**RQ3.** The measurements for RQ3 regarding reachability analysis are shown in Figure 13 with $|\Psi| \in \{0, 3, 6, 9\}$. We make the following two observations: (1) if $|\Psi|$ increases, the average runtime of the analysis also increases; (2) if $|\Psi|$ increases, the curve becomes less sharp (i.e., the measured values vary a lot more).

An explanation for (1) may be that each additional $\Psi_{\phi}$ introduces further locations and clocks thus (potentially) enlarging the state space (cf. RQ2). Observation (2) may be explained by interactions of multiple constraints thus making large parts of the state space unreachable. Consequently, adding RRCL constraints to a DSPL specification may decrease the effort for consistency checking. For instance, consider the CPU times required for the subject systems with $|\Psi| = 246$ in Figure 13. Here, reachability analysis requires less time for $|\Psi| = 9$ (red plus signs) than for $|\Psi| = 6$ (blue squares). However, in the majority of cases, composing further $\Psi_{\phi}$ increases the state space and, hence, the effort for model checking, too. Finally, note that the curves of the measurements for RQ2 and RQ3 w.r.t. deadlock freedom and liveness are shaped very similar to those in Figure 12 and 13. Only the scales of the $y$-axis significantly differ, where checking deadlock freedom is, on average, twice as fast as reachability. In contrast, liveness checks require a lot more computational effort as it consists of $|\Psi| \times |\lambda_{\Psi}|$ reachability analyses.

### 5.2 Threats to Validity

An internal threat to validity might arise from the selection of subject systems, as most of the models are synthetically generated. Due to the novelty of our approach, real-world case studies do yet not exist. However, a case study as the one presented in Weckesser et al. [29] could be adapted to evaluate our approach in a future work. Nonetheless, we expect similar results for real-world systems of comparable size (i.e., similar numbers of valid configurations) as the computational effort is mainly caused by this number. Additionally, the increasing CPU time for larger FMs (again, w.r.t. the number of valid configurations) may obstruct applicability of the approach to real-world systems. However, this issue is inherent to model checking approaches in general due to the complexity of the underlying theory (e.g., reachability checking for TA is PSPACE-hard [2]). Exploring methods for state-space reduction may be promising in future work to alleviate this issue. Moreover, not only the number, but also the type and distribution of synthetically generated constraints may significantly influence the computational effort of consistency checks. Nonetheless, generating RRCL constraints based on uniform distributions is a reasonable assumption for our assessment of the presented approach. In future work, further empirical investigations of real-world models may yield more realistic probability distributions for RRCL generation.

A threat to external validity may be the lack of comparison to other approaches. However, the reason for this is the small amount of other approaches related to DSPLs with real-time properties.

A further threat to external validity might arise from the limited expressiveness of our constraint language. We focused the design on easy usage, compositional TA construction and tractability of the consistency checks. For instance, RRCL constraints are restricted to single reconfiguration step, whereas constraints on compound reconfiguration processes are not supported. However, our compositional construction method enables us to extend the approach in various ways (up to fully-fledged TCTL formulae) in future work.

### 6 RELATED WORK

We next discuss related work regarding DSPL engineering and modeling of configurable real-time systems.

Lochau et al. [17] propose complex binding-time constraints for DSPL reconfiguration processes [10]. As a result, product (re-)configuration (including Boolean and Non-Boolean features) is modeled as an incremental (staged) process leading to complex interdependencies between features and binding times. For checking temporal consistency properties (e.g., liveness), the DSPL specification is transformed into a state-machine model which is analyzed by utilizing the model checker SPIN. To demonstrate applicability of the approach, the authors present a real-world industrial case study from the automation engineering domain (i.e., a medical device). In contrast to our approach presented in this paper, complex binding-time constraints are concerned with causal dependencies between (re-)configuration steps instead of real-time behavior.
Pfannemüller et al. [19, 20] present an adaptation logic for planning and executing reconfigurations in self-adaptive systems according to the MAPE-K loop. The knowledge component of the MAPE-K cycle is based on a context feature model. Context information is obtained by a monitoring component (e.g., by collecting sensor values) and encoded into SAT formulae. In case of multiple possible reconfigurations, the planning component estimates the costs (e.g., computational effort) for each reconfiguration and then prioritizes the presumably most efficient one. In contrast to our approach, Pfannemüller et al. neither consider real-time requirements nor use a transition system for reconfiguration processes.

Sousa et al. [26] model DSPL reconfiguration processes with labeled transition systems similar to our approach. The reconfiguration behavior is then further restricted by LTL formulae [21] thus including safety/liveness as well as constraints over arbitrary reconfiguration sequences. In addition, the approach also includes a cost function to prioritize reconfiguration decisions. Moreover, to tackle state-space explosion, the approach avoids explicit construction of the transition system, but rather operates on a symbolic representation in combination with a SAT-solver. In contrast to our model, real-time behavior is not taken into account. Nonetheless, using statistical model checking techniques may also speed up our approach (e.g., UPPAAL SMC [11]).

7 Conclusion

We presented a novel approach to specify real-time requirements for reconfiguration steps of a DSPL. We translate those feature-oriented real-time constraints into TA models in a compositional way, in order to facilitate automated consistency checks (namely deadlock freedom, reachability and liveness) for real-time critical DSPL specifications using UPFAAL. The goal of the design of the language was to achieve a suitable trade-off between ease of usage and tractability of analyses on the one hand, and a sufficient level of expressiveness on the other hand. However, due to its compositional nature, extensions to the modeling approach can be easily conducted in future work. As further future work, we plan to speed-up verification using state-space reduction and statistical model checking UPFAAL SMC. Additionally, we plan to consider further non-functional properties denoted as costs and probability information for reconfiguration steps to facilitate near-optimal reconfiguration decisions. Moreover, we plan to conduct real-world case studies such as in Weckesser et al. [29] to further evaluate applicability and usability of our approach.

Acknowledgments

This work has been funded by the German Research Foundation (DFG) as part of project A4 within the Collaborative Research Center (CRC) 1053 MAKI. This work was funded by the Hessisan LOEWE initiative within the Software-Factory 4.0 project.
REFERENCES

[1] Rajeev Alur, Costas Courcoubetis, and David Dill. 1993. Model-Checking in
Dense Real-Time. *Information and computation* 104, 1 (1993), 2–34.
[2] Rajeev Alur and David L. Dill. 1990. Automata for Modeling Real-Time Systems.
In *Automata, Languages and Programming: 17th International Colloquium War-
wick University, England, July 16–20, 1990 Proceedings*, Michael S. Paterson (Ed.).
Springer Berlin Heidelberg, 322–335. https://doi.org/10.1007/BFb0023042
[3] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. 1993. Parametric Real-
time Reasoning. In *STOC ’93*. ACM, 592–601. https://doi.org/10.1145/167888.
[4] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In
*International Conference on Software Product Lines*. Springer, 7–20.
[5] Gerd Behrmann, Alexandre David, and Kim G Larsen. 2004. A Tutorial on UPPAA
In *Formal methods for the design of real-time systems*. Springer, 200–236.
[6] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. 2006. *UPPAAL 4.0*. (2006).
[7] Mei Ya Chan, Sami Baroudi, Joseph Sui, and Jörg Liebeherr. 2018. Application-
Layer Overlay Networks for Communication-Based Train Control Systems. In
*2018 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE,
1–6.
[8] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 2012.
Behavioural Modelling and Verification of Real-Time Software Product Lines. In
*Proceedings of the 16th International Software Product Line Conference-Volume 1.*
ACM, 66–75.
[9] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2004. Staged Config-
uration Using Feature Models. In *International Conference on Software Product
Lines*. Springer, 266–283.
[10] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Beg-
sted Poulsen. 2015. *UPPAAL SMC tutorial*. International Journal on Software Tools
for Technology Transfer 17, 4 (2015), 397–415.
[11] S. Hallsteinesen, M. Hinchey, S. Park, and K. Schmid. 2015. Dynamic Software
Product Lines. *Computer* 41, 4 (April 2008), 93–95. https://doi.org/10.1109/MC.
2008.123
[12] Herman Hartmann and Tim Trew. 2008. Using Feature diagrams with Context
Reasoning. In *STOC’93*. ACM, 592–601. https://doi.org/10.1145/167888.
[13] Hans-Martin Hartmann and Tim Trew. 2008. Behavioural Modelling and Verifica-
tion of Real-Time Software Product Lines. In *Proceedings of the 16th International
Software Product Line Conference-Volume 1*. ACM, 66–75.
[14] Thomas A Henzinger, Zohar Manna, and Amir Pnueli. 1990. *The Temporal Logic
of Programs*. In *Automata, Languages and Programming: 17th International Colloquium War-
wick University, England, July 16–20, 1990 Proceedings*, Michael S. Paterson (Ed.).
Springer Berlin Heidelberg, 322–335. https://doi.org/10.1007/BFb0023042
[15] Rajeev Alur, Costas Courcoubetis, and David Dill. 1993. Model-Checking in
Dense Real-Time. *Information and computation* 104, 1 (1993), 2–34.
[16] Rajeev Alur and David L. Dill. 1990. Automata for Modeling Real-Time Systems.
In *Automata, Languages and Programming: 17th International Colloquium War-
wick University, England, July 16–20, 1990 Proceedings*, Michael S. Paterson (Ed.).
Springer Berlin Heidelberg, 322–335. https://doi.org/10.1007/BFb0023042
[17] Rajeev Alur, Costas Courcoubetis, and David Dill. 1993. Model-Checking in
Dense Real-Time. *Information and computation* 104, 1 (1993), 2–34.
[18] Lars Luthmann, Andreas Stephan, Johannes Bürdek, and Malte Lochau. 2017.
Modeling and Testing Product Lines with Unbounded Parametric Real-Time Constraints. In *Proceedings of the 21st International Systems and Software Product
Line Conference - Volume A (Sevilla, Spain) (SPLC ’17)*. ACM, New York, NY, USA,
104–113. https://doi.org/10.1145/3106195.3106204
[19] Martin Pfannemüller. 2017. *A Dynamic Software Product Line Approach for Planning and Execution of Reconfigurations in Self-Adaptive Systems*. Master’s thesis.
[20] Martin Pfannemüller, Christian Krupitzer, Markus Weckesser, and Christian
Becker. 2017. *A Dynamic Software Product Line Approach for Adaptation Plan-
ing in Autonomic Computing Systems*. In *2017 IEEE International Conference on
Autonomic Computing (ICAC)*. IEEE, 247–254.
[21] Amir Pnueli. 1977. *The Temporal Logic of Programs*. In *18th Annual Symposium
on Foundations of Computer Science (sfcs 1977)*. IEEE, 46–57.
[22] Klaas Pohl, Günter Böröcz, and Frank J van Der Linden. 2005. *Software Product
Line Engineering: Foundations, Principles and Techniques*. Springer Science &
Business Media.
[23] Carsten Saller, Malte Lochau, and Ingo Reimund. 2013. *Context-Aware DSPLs:
Model-Based Runtime Adaptation for Resource-Constrained Systems*. In *Proceed-
ing of the 17th International Software Product Line Conference co-located workshops*.
106–113.
[24] Sergio Segura, José A Galindo, David Benavides, José A Parejo, and Antonio Ruiz-
Cortés. 2012. BeTTy: Benchmarking and Testing on the Automated Analysis of
Feature Models. In *Proceedings of the Sixth International Workshop on Variability
Modeling of Software-Intensive Systems*. ACM, 63–71.
[25] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner,
Sven Apel, and Gunter Saake. 2012. *SPL Conqueror: Toward optimization of non-functional properties in software product lines*. Software Quality Journal 20, 3-4 (2012), 487–517.
[26] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. 2017. Extending
Dynamic Software Product Lines with Temporal Constraints. In *Proceedings of the
dt 2017 International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 129–139.
[27] Maurice Ter Beek, Axel Legay, Alberto Lluch Lafuente, and Andrea Vandin. 2018.
A framework for quantitative modeling and analysis of highly (re) configurable
systems. *IEEE Transactions on Software Engineering* (2018).
[28] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake,
and Thomas Leich. 2014. *FeatureIDE: An extensible framework for feature-
oriented software development*. Science of Computer Programming 79 (2014),
70–85.
[29] Markus Weckesser, Roland Kluge, Martin Pfannemüller, Michael Matthé, Andy
Schürr, and Christian Becker. 2018. Optimal Reconfiguration of Dynamic Software
Product Lines Based on Performance-Influence Models. In *Proceedings of the 22nd
International Systems and Software Product Line Conference-Volume 1*. IEEE, 98–109.
[30] Wang Yi, Paul Pettersson, and Mats Daniels. 1995. *Automatic Verification of Real-Time Communicating Systems by Constraint-Solving*. In *Formal Description Techniques VII*. Springer, 243–258.

Real-Time-Aware Reconfiguration Decisions for Dynamic Software Product Lines

SPLC ’20, October 19–23, 2020, MONTREAL, QC, Canada