Staged Configuration of Dynamic Software Product Lines with Complex Binding Time Constraints

Johannes Bürdek
TU Darmstadt
Darmstadt, Germany
johannes.buerdek@es.tu-darmstadt.de

Sascha Lity
TU Braunschweig
Braunschweig, Germany
lity@ips.cs.tu-bs.de

Malte Lochau
TU Darmstadt
Darmstadt, Germany
malte.lochau@es.tu-darmstadt.de

Markus Berens
Eckelmann AG
Wiesbaden, Germany
m.berens@eckelmann.de

Ursula Goltz
TU Braunschweig
Braunschweig, Germany
goltz@ips.cs.tu-bs.de

Andy Schürr
TU Darmstadt
Darmstadt, Germany
andy.schuerr@es.tu-darmstadt.de

ABSTRACT

Dynamic software product lines (DSPL) constitute a promising approach for developing highly-configurable, runtime-adaptive systems in a feature-oriented way. A DSPL integrates both variability in time and space in a unified conceptual framework. For this, domain features are equipped with additional binding time information to distinguish between static configuration parameters and dynamically (re-) configurable features. Until now, little support exists to specify and validate staged (re-)configuration semantics for DSPLs in a concise way. In this paper, we propose conservative extensions to domain feature models comprising variable feature binding times together with different kinds of binding time constraints. Those extensions are motivated by a real-world industrial case study from the automation engineering domain. Our implementation performs a model transformation into plain feature models treatable by corresponding state-of-the-art analysis tools. We conducted an evaluation of our approach concerning the case study.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Software Configuration Management; D.2.13 [**Software Engineering**]: Reusable Software—*Domain Engineering*; I.2.1 [**Artificial Intelligence**]: Industrial Automation

General Terms

Design, Languages, Management

Keywords

Dynamic Software Product Lines, Staged Configuration, Extended Feature Models, Industrial Case Study.

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 the author(s) 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.
VaMoS '14, January 22 - 24 2014, Sophia Antipolis, France
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2556-1/14/01 ...$15.00.
http://dx.doi.org/10.1145/2556624.2556627

1. INTRODUCTION

Software product line (SPL) engineering constitutes a comprehensive paradigm for efficiently developing similar (software) products on a common core platform [14]. An SPL implementation constitutes a generic, highly-configurable software system that is customizable to user-specific needs. The configuration space of an SPL is tailored by its supported *features*, i.e., the user-relevant product characteristics identified in the problem domain. A feature represents (1) a distinct configuration parameter to be either selected, or deselected during product derivation, as well as (2) a collection of composable engineering artifacts from the solution space to be assembled to derive the implementation variant.

SPL engineering consists of two complementary disciplines, i.e., *domain engineering* and *application engineering* [14]. During domain engineering, the relevant features within the problem space together with logical constraints restricting their possible combinations are specified in terms of a *domain feature model* [10]. The mapping of those domain features onto feature artifacts within the solution space is established by means of *variation points* within the generic SPL implementation. During application engineering, the feature model serves as a basis for product configuration, i.e., variability in the feature model is consecutively reduced until reaching a complete product configuration. Accompanying to this step-wise configuration process, the corresponding product implementation variant is automatically derived by resolving the designated variation points.

Besides extensive configuration capabilities imposing system variability *in space*, nowadays software systems further tend to exhibit variability *over time.* For instance, in the automation engineering domain, software becomes more and more prevalent being responsible for complex control and regulation tasks within production facilities with presumably long lasting life cycles [6]. Those systems must be

- *highly-configurable* prior to their initial activation to be customizable to varying arrangements of the production plant, as well as
- *adaptive*, i.e., adjustable to changing operational situations at runtime throughout the product life-cycle.

To cope with those different kinds of system diversity in a common conceptual framework, *dynamic software product lines* (DSPLs) [6] constitute a natural generalization of SPLs integrating both variability in space *and* over time into a unified product line specification. Feature *binding times* are introduced to distinguish (1) *static* features denoting (pre-)configuration decisions prior to product implementation derivation and (2) *dynamic* features referring to left-open variation points within those implementations to enable (re-)configurations at runtime.

The resulting *staged configuration* for DSPL product variant derivation partitions the set of feature configuration steps into consecutive *stages* according to their binding times. This way, additional *temporal* dependencies among feature selections arise as stages impose an implicit prioritization among configuration choices. After having configured all static features, the resulting (partial) product implementation still contains variation points for the remaining dynamic features thus offering several implementation variants at runtime. Hence, variation points are preserved as first-class entities even in final product implementations.

Summarizing, engineering a DSPL requires capabilities to (1) specify and validate binding times and corresponding constraints during domain engineering and (2) conduct staged configurations during application engineering as well as reconfigurations at runtime. Until now, no comprehensive and generally accepted approach exists for DSPL engineering that allows for a systematic specification and analysis of feature binding times. To tackle those deficiencies, we propose the following enhancements to (extended) domain feature models with feature attributes [11].

1. Assignment of (multiple) binding times to features.
2. Specification of complex constraints including dependencies between binding times, feature selections and/ or feature attribute values.

All concepts are motivated by a real-world case study provided by our industrial partner Eckelmann AG[1] from the automation engineering domain, i.e., a DSPL comprising configurable, adaptive *Device Control Units (DCU)* for radiotherapy. The semantics of our feature model extensions are defined by a model transformation into a plain feature model. This allows us to apply existing modeling, analysis and configuration capabilities for SPLs also to DSPL engineering as demonstrated by our DCU DSPL case study.

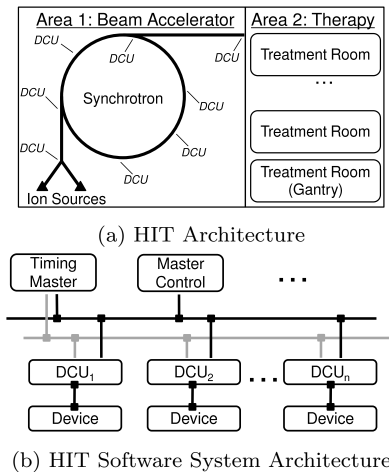
2. DSPL CASE STUDY

Our research on DSPL engineering is motivated by a real-world industrial case study provided by our industrial partner Eckelmann AG. We first give a technical overview of the case study focusing on those properties and domain-specific characteristics relevant for re-engineering it as a DSPL.

2.1 The DCU Case Study

Our case study consists of the software system of a complex medical device in the Heidelberg Ion-Beam Therapy Center (HIT) [7] which has been developed and implemented by Eckelmann AG. The HIT provides a novel treatment for radiotherapy. In contrast to conventional radiotherapy, tumors are treated with heavy ion beams. This new technique has crucial advantages as it enables the irradiation of deeply positioned tumors and a very precise irradiation beam positioning which reduces the damages of unaffected human tissues located near by the tumor.

[1]http://www.eckelmann.de/en



(a) HIT Architecture

(b) HIT Software System Architecture

Figure 1: System and Software Architecture of HIT

The architecture of the HIT is shown in Fig. 1 comprising (1) the beam accelerator area and (2) the therapy area. In (1) the ion sources and the synchrotron are located being responsible for the ion beam creation. Area (2) contains a variable number of treatment rooms of different types.

For beam creation, several physical devices, e.g., magnets as well as E/E devices are installed around the synchrotron. Here, we focus on the software components of the HIT automating the control cycle of the therapy process thus constituting an adaptable real-time/safety-critical embedded software system. As depicted in Fig. 1, the overall software system for coordinating the devices during the therapy process is distributed over designated *Device Control Units* (DCUs). For instance, one DCU is responsible for controlling the angle position and the intensity of the magnets adjusting the beam cycle. The different DCUs are integrated via a real time bus (RTB) constituting a hierarchical master-slave architecture with a timing master for synchronizing time-critical tasks (cf. Fig. 1b). In each control cycle, the master control initiates the beam creation, sets up the beam control and type-specific DCU parameters, continuously supervises the DCU status and controls the DCU operation mode at runtime. Each DCU receives type-specific parameters from the master control after activation. Those type-specific parameters are not fixed, but may change throughout control cycles. After initialization, the master control sets up the beam control parameters for each DCU. The behavior and runtime adaptivity of each DCU is mainly determined by its *operation mode*, e.g.,

- in the *manual mode* every DCU is fully maintainable, e.g., for RTB calibration,
- in the *therapy mode* each DCU strictly accomplishes a predefined procedure, whereas
- in the *experiment mode* every type-specific DCU parameter is freely (re-)configurable and
- in the *idle mode* the system is suspended.

The current operation mode further restricts DCU parameter (re-)configurability at runtime due to safety reasons, e.g., in *thearapy mode* an incorrect (re-)configuration may endanger the patient's health. The actual runtime behav-

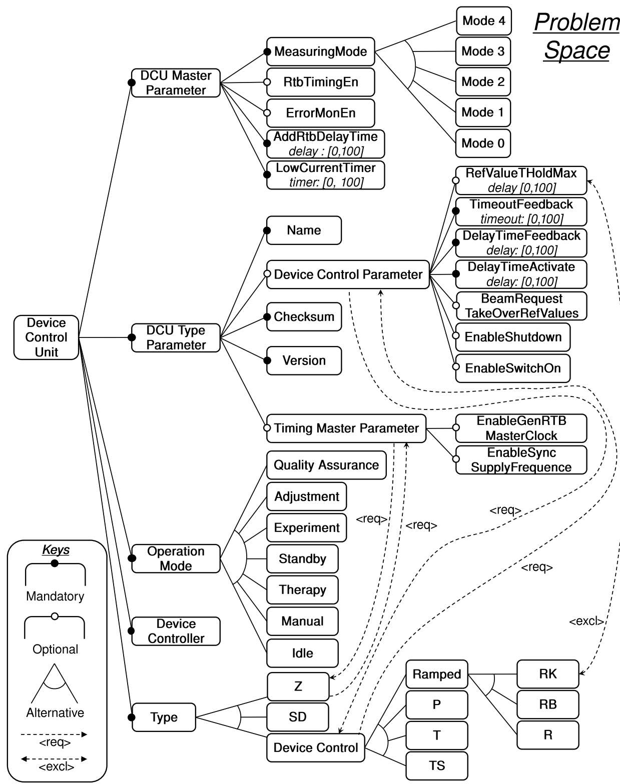


Figure 2: Domain Feature Model of the DCU DSPL

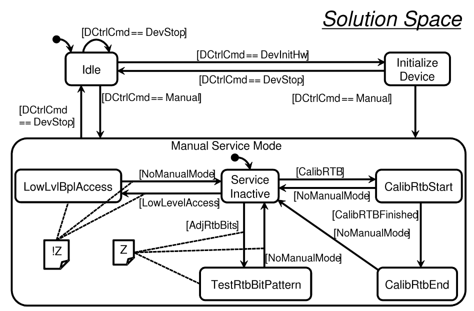


Figure 3: DCU DSPL Solution Space Artifacts

ior of each DCU consists of continuously setting and updating control parameters of its designated device depending on the progress of the overall therapy procedure, its parametrization and current status. Hence, the software of the different DCU types constitutes a family of similar, yet well-distinguished, highly-configurable and adaptive control systems with complex interdependencies between configuration parameters and/or respective implementation artifacts.

2.2 Re-Engineering the DCU as DSPL

The characteristics of the DCU described in the previous section strongly indicate its predestination to be re-engineered as a (D)SPL. SPL engineering distinguishes between *domain engineering* and *application engineering* [14].

During domain engineering, the system is decomposed into its set of features. Those features are usually organized in a tree-like (domain) *feature model* to define dependencies and constraints among feature combinations. We identified about 100 parameters constituting domain features being relevant for DCU configuration thus resulting in a configuration space with many more than 100,000 DCU variants. We focus our considerations on an extract of the complete DCU feature model comprising 47 features (cf. Fig. 2). In addition to Boolean features, the DCU DSPL further offers non-Boolean configuration choices by means of feature attributes [11]. For instance, the feature *DelayTimeActivate* has an attribute *delay* of $[0 \dots 100]$ milli-seconds.

As a second step, engineering artifacts are designed that allow for an automated assembling of an implementation variant for each valid configuration. For this, a mapping of domain features onto corresponding artifacts is established throughout all abstraction levels, i.e., from requirements down to code. For instance, during domain engineering of the DCU case study we consider artifacts occurring as part of a model-based development process, e.g., state machine models. We apply a variability modeling approach using selection conditions over feature parameters as illustrated in the state machine extract in Fig. 3. The state machine model integrates all artifacts of every possible model variant. Starting in the *idle* mode, a DCU may switch to the *hardware initialization* mode and/or the *manual service* mode (if selected in a configuration). A DCU of type Z (timing master DCU) has further timing parameters ($rtbBitPattern$) adjustable and tested in the manual mode.

Product variants are derived from an SPL by configurations, i.e., by selecting a set of features from a feature model, whereas the remaining ones are deselected. This step-wise configuration process is referred to as *staged configuration* [5], where in each stage a corresponding subset of features is considered. The separation of features into stages is indicated by their *binding times* also imposing a temporal ordering among the stages. We distinguish between *static* and *dynamic* binding times. Features with static binding times denote (pre-)configuration decisions prior to product derivation and are, therefore, immutable, e.g., the DCU type. In contrast, dynamic features, e.g., the DCU operation mode, are (re-)configurable for already derived products allowing preplanned runtime adaptation. The ordering of static binding times is strict, whereas stages of dynamic binding times may be re-entered for runtime reconfiguration.

As shown in Fig. 3, the solution space artifacts corresponding to static features are parameterized by means of feature-annotations for projecting only those artifacts being relevant for a particular product implementation. For instance, the state *TestRtbBitPattern* is responsible for the validation of the timing master parameters thus being annotated with *Z*. In contrast, variation points controlled by dynamic features are preserved in a DSPL product implementation, e.g., encoded via transition guards like *[DCtrlCmd == Manual]* in Fig. 3, in order to allow for a (de-)activation of artifacts affected by a runtime reconfiguration. Hence, DSPL implementations consists of an interleaving of conventional control logic artifacts and variation points encoded at the same level of abstraction thus imposing complex *logical* as well as *temporal* constraints among those entities and their (re)configuration order [15].

However, recent feature modeling approaches do not allow for the specification and validation of logical and temporal

constraints involving features, attributes and their binding times in a configuration process. To tackle these drawbacks, we extend feature models with binding time constraints to restrict DSPL (re-)configuration processes in order to make DSPLs reliably applicable for safety/real-time critical application domains as given by our case study.

3. FEATURE MODELS WITH BINDING TIME CONSTRAINTS

3.1 Extended Feature Models

Feature models provide a formal basis for specifying the configuration space of an SPL. A feature model consists of

- a finite set F of *feature names* and
- a finite set Φ of *dependencies* between features in F.

According to Batory, it is sufficient to represent a feature model by interpreting F as a set of Boolean variables and Φ as a propositional formula over F [1].

However, to support intuitive modeling of SPL configuration spaces during domain engineering, feature models also provide a graphical layout. Here, we consider FODA feature diagrams [10] as shown in Fig. 2. Feature diagrams organize the set F of features in a tree-like structure defining a feature *decomposition* relation $f \prec f'$, i.e., a hierarchical dependency between a parent feature f and its child features f'. Thereupon, various constructs have been proposed to express further kinds of constraints among features.

- Feature modalities to distinguish *mandatory* and *optional* child features of a parent feature, e.g., a *Device_Controller* is a mandatory part of each DCU variant, whereas the *EnableShutdown* function is optional.
- Feature groups define combinatorial constraints on sets of sibling features, e.g., the operation modes of a DCU define an *alternative* group to guarantee that a DCU constantly operates in one specific mode.
- Cross-tree constraints, i.e., binary require-edges $f \rightarrow f'$ and exclude-edges $\neg f \rightarrow f'$, e.g., the feature *Device_Control* requires *Device_Control_Parameter* except for DCUs of type Z.

We further distinguish *concrete* and *abstract* features [17]. The former define user-visible configuration parameters as usual, whereas the latter just serve as additional structuring elements within the feature tree.

Further extensions to FODA-like feature models were proposed which are usually collected under the notion of *extended feature models* (EFM) [13]. We consider as one extension *non-Boolean* features, i.e., features f may be equipped with *feature attributes*, referred to as $f.a$. Feature attributes correspond to typed variables denoting additional properties configurable for feature f.

For instance, the attribute $AddRtbDelayTime.delay$ is used to define a position-specific synchronization delay value for the timing master (cf. Fig. 2). We limit our considerations to attributes typed over finite value domains T, namely enumerations and integer intervals. Accordingly, we enhance cross-tree constraints to also incorporate conditions over values of feature attributes. We generalize the aforementioned cross-tree constraints to

$$LHS \Rightarrow RHS$$

where the left-hand-side (LHS) and/or the right-hand-side (RHS) may be negated to define exclude-constraints. Besides features f, LHS as well as RHS may refer to values of feature attributes $f.a$ by means of expressions

$$f.a \ relop \ c, \text{ where } relop \in \{=, \neq, \leq, \geq, <, >\}$$

and $c \in T$ being a constant value. Those constraints allow, e.g., to express dependencies between timeout attributes of time-critical features. Experiences from our case study have shown us that those kinds of constraints are expressive enough to capture crucial *logical dependencies* arising among features and/or attributes of the DCU DSPL. However, those constraints do not suffice to express *temporal constraints* among configuration choices during staged configuration and runtime reconfiguration.

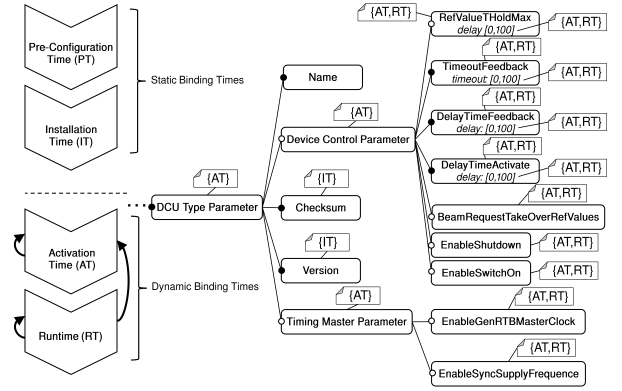


Figure 4: Feature Binding Times of the DCU DSPL

3.2 Adding Binding Times to Feature Models

Recent DSPL approaches specify orderings among configuration steps by assigning *binding times* to features [6]. This way, the configuration process is separated into consecutive *stages*, each corresponding to a binding time.

However, in addition to those binding time assignments we observed in our case study the need for richer constructs to concisely specify the staged configuration semantics of a DSPL. In particular, the DCU DSPL incorporates

1. features/attributes with multiple possible binding times dynamically decided and restricted during application engineering and
2. complex constraints between feature/attribute configuration choices and binding times.

We identified several cases in the DCU DSPL where features as well as attribute values may be either selected/deselected in some stage, or this decision might be delayed to some later stage. To integrate binding time information for every feature f into the feature model, we consider binding times to constitute a special kind of attribute denoted $f.bt$. Those binding time attributes are typed over a subset $BT' \subseteq BT$ of the enumeration type BT defining all binding times of the DSPL. BT is further equipped with

- a total ordering denoting the sequencing of the corresponding configuration stages and
- a distinction between static and dynamic binding times.

If no explicit binding time declaration is annotated to a feature/attribute, we assume as *default* binding time the entire set BT. Fig. 4 shows an extract of the DCU feature model with $BT = \{$*Pre-configuration Time*, *Installation Time*, *Activation Time*, *Runtime*$\}$ where *Pre-configuration Time* and *Installation Time* are static, whereas *Activation Time* and

	Feature/Attribute	Binding Time
Feature/ Attribute	1. Feature/Attribute Constraints as usual	2. Configuration Decision restricts Binding Time
Binding Time	3. Binding Time restricts Configuration Decision	4. Dependencies among Binding Times

Table 1: Four Kinds of Binding Time Constraints

Runtime are dynamic binding times. Hence, the *Version* is to be configured at installation time, whereas *DelayTimeFeedback* is (re-)configurable at activation time and runtime and the *Name* may be (re-)set in any stage.

Each configuration decision potentially influences subsequent configuration options within the same and/or subsequent stages. Those influences are not only imposed by the logical constraints in Φ, but also by the temporal ordering of those decisions in a staged configuration and due to runtime reconfigurations. For instance, the feature *Quality_Assurance* affects *DelayTimeActivate* and its attribute *delay* as it must be configured in the same stage as *Quality_Assurance* in order to allocate corresponding RTB capabilities. Thus, a complex interplay of feature/attribute dependencies and their binding times arise potentially inducing unintended interactions among both. This becomes even worse in case of (reconfigurable) DSPLs as conventional program logics, feature parameters and their related artifacts arbitrarily interleave at implementation level (cf. Fig. 3).

A natural extension to feature/attribute constraints defined by a feature model seems to also allow dependencies between feature/attributes and feature binding times both as part of the right-hand-side as well as left-hand-side. Thus, four kinds of constraints including features/attributes and feature binding times arise as summarized in Table 1 with rows denoting *LHS* and columns denoting *RHS*.

Similar to feature constraints, binding time constraints have various sources, e.g., requirements, technical reasons, legal restrictions etc. In the DCU DSPL, we identified all kinds of constraints involving binding times.

- *Therapy ⇒ (EnableSyncSupplyFrequence.bt = Activation_Time)* – The operation mode *Therapy* requires the configuration of *EnableSyncSupplyFrequence* at activation time to prevent a patient to be harmed by an adaptation during radiotherapy (cf. Table 1: Case 2).
- *(DelayTimeActivate.bt = Runtime) ⇒ Experiment* – Reconfiguring the *DelayTimeActivate* at runtime is only permitted in the *Experiment* mode since it may causes deaths in other modes (cf. Table 1: Case 3).
- *(DelayTimeFeedback.bt = Activation_Time) ⇔ (TimeoutFeedback.bt = Activation_Time)* – Configuring *DelayTimeFeedback* at activation time requires to configure *TimeoutFeedback* in the same stage and vice versa in order to ensure that both are bound at *activation time* (cf. Table 1: Case 4).

However, integrating those temporal aspects into feature models not only allows for a precise definition of DSPL (re-)configuration semantics, but also introduces new potential sources for invalid specifications. Various kinds of *anomalies* in feature models and corresponding analysis techniques were already proposed in the literature [2] that are also adoptable to EFMs with binding time constraints.

- *Void Feature Models:* No valid complete product configuration exists satisfying Φ.
- *Core Features/Attribute Values/Binding Times:* Due to logical as well as binding time constraints, certain features, feature attribute values and/or binding times are part of every valid (staged) configuration although not being declared mandatory.
- *Dead Features/Attribute Values/Binding Times:* Dead features/attribute values and dead binding times are never selectable for any valid (staged) configuration as they are excluded by the constraints.

To rule out those unintended cases already during domain engineering for ensuring correct staged configuration/reconfigurations during application engineering and runtime, binding-time-aware feature model validation approaches are required. Thereupon, staged configuration engines are to be extended, accordingly, to also take binding time constraints into account, e.g., to avoid potential deadlocks.

4. FEATURE BINDING TIME ANALYSIS

Enriching feature models with binding time constraints requires for corresponding enhancements of existing approaches for (1) configuration space validation during domain engineering and (2) continuous consistency checking of staged (re-)configuration steps. To support both we define a precise evaluation semantics of our feature model extensions by means of a transformation into plain feature models.

4.1 Model Transformation

We propose a translation of feature models with attributes and complex binding time constraints into plain feature models solely containing *Boolean* features and corresponding logical constraints. The resulting plain feature model allows for applying state-of-the-art constraint solving capabilities. The overall translation scheme is illustrated in Fig. 5a performing three consecutive steps, i.e., (1) feature attribute transformation, (2) binding time integration and (3) constraint translation. The resulting plain feature model consists of two sub-trees, i.e., the original feature tree *FM* with flattened attributes and a binding time tree *BT* with the binding time information for features and attributes.

Feature Attribute Transformation.

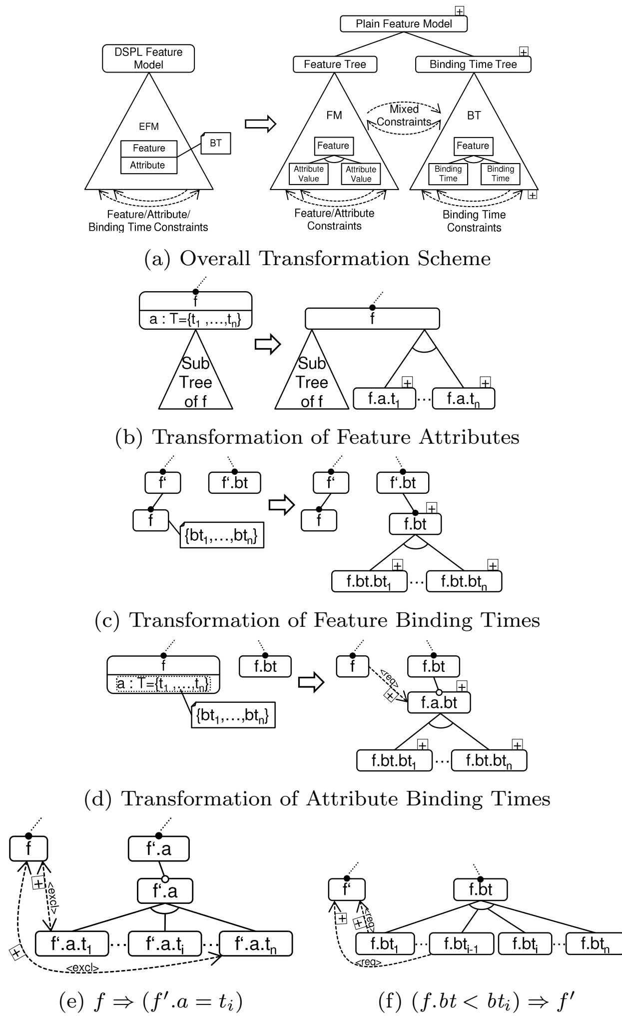
Feature attributes $f.a$ of type T, intrinsically belong to their features f and, therefore, have to receive a definite value $v \in T$ whenever f is selected in a configuration. Otherwise, if feature f is deselected, all its attributes $f.a$ are also absent. Hence, each feature attribute $f.a$ may be seen as a mandatory, non-Boolean sub feature of f. As we limit our considerations to attributes with finite value domains, we are able to enumerate any value $v \in T$ assignable to $f.a$ into a fresh alternative group underneath f. The corresponding transformation rule in Fig. 5b is applied to each attribute $f.a_i$ of all features $f \in F$. The parts annotated with + represent new elements added by the rule.

Feature Binding Time Integration.

Binding times $f.bt$ of features f constitute a special kind of attribute of f with type $BT' \subseteq BT$. However, two main differences arise, namely

- their values are *implicitly* assigned during staged configuration w.r.t. the stage in which f is configured and
- they always receive a value regardless of whether f is selected or deselected.

Thus, the transformation of feature binding times slightly



(a) Overall Transformation Scheme

(b) Transformation of Feature Attributes

(c) Transformation of Feature Binding Times

(d) Transformation of Attribute Binding Times

(e) $f \Rightarrow (f'.a = t_i)$

(f) $(f.bt < bt_i) \Rightarrow f'$

Figure 5: Transformation Rules

differs from those of feature attributes. As depicted in Fig. 5c, we introduce an orthogonal sub tree to organize the binding time information. Therein, each feature f from the original feature model receives a dedicated (abstract) mandatory feature $f.bt$ serving as binding time parameter for f. This sub tree duplicates the feature hierarchy $\prec$ of the original feature tree only for readability reasons. The sub set BT' of binding times declared for f forms a corresponding alternative group underneath $f.bt$ similar to the attribute transformation. This transformation does not induce any dependency between feature f and its binding time feature $f.bt$. In contrast, those relationships are enforced by adapting the staged configuration semantics, accordingly.

Attribute Binding Time Integration.

Feature attribute binding times differ from those of features due to the close relationship between attributes and their associated features. Various semantic interpretations may be applied especially concerning the case where a feature f with attribute $f.a$ is deselected, e.g., (1) $f.a$ inherits the binding time of f in case f is deselected, (2) $f.a$ does not exhibit any binding time if f is deselected, or (3) the binding times of $f.a$ and f are independent. Here, we consider a compromise between those interpretations reflecting the temporal aspects of a staged configuration.

- if f is deselected before $f.a$ has been configured, $f.a$ has no binding time value in the final configuration.
- if f is deselected after $f.a$ has been configured, $f.a$ has some binding time value in the final configuration.

This corresponds to the intuition that constraints on attribute values (as well as their binding times) are only claimed if the related feature is selected (see below). The corresponding transformation rule is depicted in Fig. 5d. For each attribute $f.a$ of feature f, an abstract optional binding time feature $f.a.bt$ with a corresponding alternative group for possible binding time values is added underneath $f.bt$. To enforce a binding time value for each attribute of a selected feature, a corresponding require edge is added.

Constraint Translation.

The transformation rules in Fig. 5e and Fig. 5f exemplify the translation of constraint expressions. If RHS consists of a constraint over attribute values ($f'.a$ in Fig. 5e), the constraint is translated using exclude-edges from LHS (f in Fig. 5e) to all invalid values, rather than a require-edge to the valid value t_i. Otherwise, the selection of f would enforce the selection f' which is not the intended semantics. If RHS consists of a feature (f' in Fig. 5f), all selections satisfying LHS (binding times of f prior to bt_i in Fig. 5f) require the selection of that feature. Other combinations of those cases of LHS and RHS are translated, accordingly. A valid configuration C of a transformed feature model comprises

- a selection/deselection decision, denoted $\langle +f \rangle / \langle -f \rangle$, together with the respective binding time $\langle f.bt := bt_i \rangle$ for *each* feature $f \in F$ and
- a feature attribute value $\langle f.a := a.t_i \rangle$ assignment together with its respective binding time $\langle f.a.bt := bt_j \rangle$ for those attributes assigned to selected features

satisfying the translated set of constraints Φ.

In SPL domain engineering, feature model anomalies, e.g., unsatisfiable constraints, are major subjects to validation concerns [2] as they potentially obstruct (staged) configuration semantics [9]. Thereupon, the additional temporal dimension of this problem arising for DSPLs due to complex binding time constraints lead to new potentials of invalidity during domain engineering. Our proposed integration of those constraints into feature models now helps us to formally verify the absence of those anomalies prior to DSPL application derivation and runtime reconfiguration.

4.2 Validation of Binding Time Constraints

As a major requirement for the validity of SPL configuration space specifications defined by a feature model, we consider (1) its *satisfiability*, i.e., at least one valid product configuration C exists that satisfies all constraints and (2) the absence of unattended *core* features and *dead* features, i.e., every feature is actually selectable as well as deselectable in a staged configuration. The adoption of those properties to feature models with binding time information means to take binding time constraints into account.

Thus, *binding-time-aware* satisfiability holds iff (a) the constraints between features and/or attributes are satisfiable as usual and (b) the binding time constraints do not lead to configuration decisions with no valid binding time left. Due to our transformation described above both (a)

and (b) hold iff the plain feature model is satisfiable. Correspondingly, *binding-time-aware* dead/core feature analyses require (a) the absence of feature and/or attribute values that are not selectable/deselectable and (b) the absence of feature binding times that are never enabled. Again, our transformation ensures both cases to be reducible to a constraint solving problem on plain feature models [2].

In the DCU example, the configuration order of the attributes *TimeoutFeedback.timeout* and *DelayTimeFeedback.delay* is crucial to yield a valid configuration. Both features and their attributes are assigned to the stages *activation time* and *runtime* and are, therefore, potentially configurable at different binding times. Nevertheless, the value of *TimeoutFeedback.timeout* depends on the value of *DelayTimeFeedback.delay* thus the configuration of *DelayTimeFeedback.delay* must precede *TimeoutFeedback.timeout*.

Those validations during domain engineering ensure that only meaningful (re-)configurations are permitted for the respective DSPL variants. However, for every step during staged (re-)configuration the *logical* as well as *temporal* constraints among open configuration choices have to remain satisfiable in subsequent stages.

4.3 Staged Configuration

The validity checks performed during domain engineering concerning the satisfiability of a feature model ensure that at least one complete product configuration C exists. During application engineering, the configuration C is derived from a feature model *FM* in a step-wise way in a sequence

$$C_0, C_1, \ldots, C_n$$

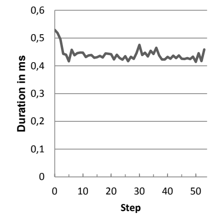
of consecutive *partial* configurations C_i with open configuration decisions. Every intermediate configuration C_i *refines* its preceding configuration C_{i-1} by (1) preserving the decisions of C_i and (2) adding further decisions made in that step being consistent to the previous decision w.r.t. Φ.

For instance, the configuration decisions $\langle +Device_Control_Unit\rangle$, $\langle +Device_Controller\rangle$, $\langle +Type\rangle$, $\langle +Z\rangle$,... yield a sequence $C_0, C_1, C_2, C_3, \ldots$ of valid partial configurations. To guarantee those sequences to eventually terminate in a valid complete configuration $C_n = C$, it has to be ensured that the constraints in Φ remain satisfiable after each configuration step [9]. For feature models with binding time information, those configuration sequences are further restricted to *staged configurations* satisfying the binding time constraints. Again, the satisfiability check performed during domain engineering solely ensures that there exists a temporal ordering of configuration steps (1) leading to *some* valid product configuration C and (2) complying with the constraints. Besides selection decisions for features f and attribute values $f.a$, a configuration C now also comprises binding times $f.bt$ and $f.a.bt$ for those selections. Hence, a configuration step from a partial configuration C_i to a subsequent partial configuration C_{i+1} now consists of a pair of a feature/attribute selection/deselection decision and the binding time determined by the current configuration stage. For our previous example, we now have, e.g., the pair $(\langle +DCU\rangle, \langle DCU.bt := Pre\text{-}Configuration\ Time\rangle)$, as first configuration step (where *DCU* abbreviates *Device Control Unit*).

Due to our transformation into plain feature models, both constraints on features/attributes and binding times are tractable equally when continuously verifying satisfiability during staged configuration. In addition, it has to be ensured

	EFM	FM
#Features	47	783
#Constraints	55	61
#Core Features	6	
#Core BT	39	
#Dead Features	0	
#Dead BT	0	
SAT Check	0.46ms	

(a) Domain Engineering Analysis



(b) Staged Configuration SAT Check

Figure 6: Results of the Evaluation

that the sequence of configuration decisions complies to the ordering defined for the set of binding times, i.e., sequences $C_0, C_1, \ldots, C_n$ are restricted such that (1) they constitute a correct overall ordering of the binding times w.r.t. the stage order and (2) each configuration decision eventually takes place in some of its designated stages. To ensure this, the set of binding times left for not yet decided features/attributes is to be continuously reduced, i.e., (implicitly) deselected accompanying to the progress of the staged configuration.

For example, assume the configuration of *DCU_Master_Parameter* in a step C_i to be the first decision to take place during *installation time*. Thus, the set of remaining binding times for all open configuration decisions is (implicitly) reduced by deselecting *pre-configuration time* after step C_i.

5. IMPLEMENTATION AND EVALUATION

We developed an implementation of the presented feature model transformation to evaluate the binding time analyses approach w.r.t. the DCU case study. We first transform an EFM with binding times into a plain feature model both based on an EMF meta model which allows us to apply off-the-shelf solver tools for feature model analysis. The transformation rules in Sect. 4 are implemented with eMoflon, a CASE tool for Model-Driven Software Development[2]. eMoflon further provides Story Driven Modeling (SDM) to define the control flow for applying graph transformation rules. The transformation is used for experimental domain engineering analyses as well as staged configuration simulations. To provide scalable binding time analysis especially between subsequent staged configuration steps, we applied a very efficient constraint solver optimized for runtime adaption of mobile devices modeled as a DSPL [16].

The analyses were performed by a Windows 7 with i5-3230M machine (2.6GHz) processor and 16GB of RAM. Table 6a provides a comparison of the EFM of the DCU and the resulting plain FM. The additional features in the FM result from the transformation of attributes and binding times to Boolean features, e.g., several attributes have value ranges from 0 to 100, e.g., the *timeout*. The satisfiability check takes 0.46ms and has shown that the DCU feature model remains consistent after being enriched with binding time constraints. We further injected anomalies, e.g., dead features with no valid binding left which has also been detected efficiently. We also evaluated application engineering scenarios with our approach by simulating staged configuration sequences. After each step a satisifiability check was applied to avoid deadlocks in subsequent stages. Fig. 6b depicts the development of the satisfiability check duration after each

[2]http://www.moflon.org/

configuration step that constantly requires 0.45ms independently from the number of bound features/attributes in the current partial configuration. The complete configuration process (53 steps) took 23.5ms. We, again, injected errors to provoke unsatisfiable partial configurations that were successfully detected requiring 0.45ms. Summarizing, the high efficiency of validity checks obtained for an industrial-size example makes our approach a feasible extension to recent staged configuration engines.

6. RELATED WORK

DSPLs were introduced in [6] including the idea of feature binding times. Mei et al. [12] propose a feature-oriented domain modeling method including binding times but mainly concentrate on their graphical layout. Rosenmüller et al. [15] describe the integration of static and dynamic features in SPL implementations similar to our state machine example in Fig. 3 to enable flexible feature binding times. Both approaches only allow for a singleton binding time per feature, whereas our approach permits multiple feature/attribute binding times together with binding time constraints.

In [2], Benavides et al. propose to transform EFMs into a constrained satisfaction problem (CSP) for analyses purposes. Thereupon, Karataş et al. [11] present a translation of EFMs with attribute constraints into constraint logic programming over finite domains thus enabling the application of standard constraint solvers to analyze EFMs. White et al. [18] extend those works to allow the analysis of multi-step feature models, i.e., the analysis of partial configurations.

The constraint languages considered in these publications are usually more expressive than the one required for a case study and are, therefore, not transformable into plain feature models as done in our implementation. In addition, those analysis techniques do not support mixtures of logical and temporal constraints as in our approach.

Various authors present formal semantics for staged configuration of feature models, e.g., using a combination with multi-views [4] or multi-levels [3], respectively, and workflow modeling [9]. All those approaches rely on an additional model separated from the domain feature model thus separating configuration choices from binding times.

7. CONCLUSION AND FUTURE WORK

In this paper, we proposed extensions to domain feature models to support formal modeling and analysis of DSPL staged configuration semantics with complex binding time constraints. Those extensions were motivated by an industrial case study, i.e., a DSPL from the automation engineering domain provided by our industrial partner. Our sample implementation is based on a model transformation and we conducted an evaluation w.r.t. our case study. As a future work, we plan to investigate even more expressive constraints as well as feature attributes with unrestricted value domains [11]. To validate binding time constraints during reconfiguration processes at runtime, we plan to combine our model with an automata-based reconfiguration model [8]. Our goal is to integrate those approaches into a binding-time-aware safety-critical DSPL adaptation engine.

8. ACKNOWLEDGMENTS

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP 1593: Design For Future – Managed Software Evolution.

9. REFERENCES

[1] D. Batory. Feature models, grammars, and propositional formulas. In *SPLC'05*, pages 7–20, 2005.

[2] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *CAiSE*, pages 491–503, 2005.

[3] A. Classen, A. Hubaux, and P. Heymans. A Formal Semantics for Multi-level Staged Configuration. In *VaMoS'09*, pages 51–60, 2009.

[4] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[5] K. Czarnecki, S. Helsen, and E. Ulrich. Staged Configuration Using Feature Models. In *SPLC'04*, pages 266–283, 2004.

[6] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41:93–95, 2008.

[7] Heidelberg University Hospital. Heidelberg Ion-Beam Therapy Center. http://www.klinikum.uni-heidelberg.de/, 2013.

[8] M. Helvensteijn. Dynamic delta modeling. In *SPLC'12*, pages 127–134. ACM, 2012.

[9] A. Hubaux, A. Classen, and P. Heymans. Formal modelling of feature configuration workflows. In *SPLC'09*, pages 221–230, 2009.

[10] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and S. A. Peterson. Feature Oriented Domain Analysis (FODA). Technical report, CMU, 1990.

[11] A. S. Karataş, H. Oğuztüzün, and A. Doğru. Mapping extended feature models to constraint logic programming over finite domains. In *SPLC'10*, pages 286–299. Springer, 2010.

[12] H. Mei, W. Zhang, and F. Gu. A feature oriented approach to modeling and reusing requirements of software product lines. In *COMPSAC '03*, 2003.

[13] L. T. Passos, T. Berger, M. Novakovic, K. Czarnecki, Y. Xiong, and A. Wasowski. A study of non-boolean constraints in variability models of an embedded operating system. In *SPLC'11*, pages 21–28, 2011.

[14] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[15] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Code generation to support static and dynamic composition of software product lines. In *7th GPCE*, pages 3–12. ACM, 2008.

[16] K. Saller, M. Lochau, and I. Reimund. Context-aware dspls: model-based runtime adaptation for resource-constrained systems. In *SPLC'13*, pages 106–113. ACM, 2013.

[17] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. Abstract features in feature modeling. In *In SPLC'11*, pages 191–200, 2011.

[18] J. White, B. Dougherty, D. C. Schmidt, and D. Benavides. Automated reasoning for multi-step feature model configuration problems. In *SPLC'09*, pages 11–20, 2009.