CardyGAn: Tool Support for Cardinality-based Feature Models

Thomas Schnabel
TU Darmstadt
thomas.schnabel@stud.tu-darmstadt.de

Markus Weckesser
TU Darmstadt
markus.weckesser@es.tu-darmstadt.de

Roland Kluge
TU Darmstadt
roland.kluge@es.tu-darmstadt.de

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

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

ABSTRACT

Cardinality-based feature models (CFM) constitute a crucial and non-trivial extension to FODA feature models in terms of UML-like feature multiplicities and corresponding cardinality constraints. CFM allow for specifying configuration choices of software systems incorporating multiple instances (copies) of features, e.g., for tailoring customer-specific and even potentially unrestricted application resources. Nevertheless, the improved expressiveness of CFM compared to FODA feature models complicates configuration semantics, including sub-tree cloning and potentially unbounded configuration spaces. As a consequence, entirely novel anomalies might arise such as dead cardinality intervals, false unboundedness, and cardinality gaps, which are not properly treated by recent feature-modeling tools. In this paper, we present comprehensive tool support for assisting specification, validation, and configuration of CFM. Our tool CardyGAn, therefore, incorporates capabilities for CFM editing, automated CFM validation including anomaly detection based on a combination of ILP and SMT solvers, as well as a CFM configuration engine based on Alloy.

CCS Concepts

•**Computing methodologies** → *Model verification and validation;* •**Software and its engineering** → **Domain specific languages; Software configuration management and version control systems;** Software product lines;

Keywords

Extended Feature Models, Automated Validation.

1. INTRODUCTION

Cardinality-based feature models (CFM) constitute a conservative, yet non-trivial extension to FODA feature diagrams [15] by means of UML-like *feature multiplicities* [22], represented as *cardinality annotations* with corresponding cardinality-interval constraints [10]. In this regard, CFM allow for selecting multiple instances (*copies*) of the same *type* of feature within a configuration, including (recursive) *clones* of the corresponding feature sub-trees [9]. In a similar way, the further modeling constructs known from the FODA notation, namely feature groups and cross-tree edges, are enhanced with feature cardinality constraints, too. Those extended variability-modeling concepts of CFM are of particular relevance for specifying configuration spaces of many of today's emerging application domains such as communication networks and multi-agent systems [20]. In particular, for those kinds of systems, not only the *type*, but also the *amount* of available components, resources etc. is supposed to be explicitly configurable by the user, especially including (virtually) unrestricted (*unbounded*) resources as apparent, e.g., in cloud computing [23, 20].

Nevertheless, the improved expressiveness of CFM compared to FODA feature diagrams does not come for free. The different kinds of cardinality annotations and constraints, and their possible interplay within models of realistic sizes not only complicates graphical syntax but also configuration semantics of CFM compared to FODA feature diagrams [9, 17, 19, 6]. Inconsistencies among cardinality-interval constraints potentially lead to entirely novel kinds of anomalies in CFM, e.g., dead cardinality [18], false unbounded cardinality intervals, and cardinality interval gaps. Constraint solvers recently used for FODA feature diagram analysis, e.g., SAT [16] and CSP [4], are inappropriate for automated CFM validation, mainly due to their lack of ability to handle potentially infinite configuration spaces. As a consequence, comprehensive tool support for SPL engineering incorporating CFM is needed, accompanying both (1) domain engineering and (2) application engineering [5], in order for CFM to finally become tractable and established in practice. To this end, tool support for CFM must include (1) CFM editing assistance together with capabilities for automated CFM validation and anomaly detection, and (2) mechanisms for (staged) configuration of a given CFM.

In this paper, we present our CFM tool CardyGAn[1],

VaMoS '16, January 27-29, 2016, Salvador, Brazil
© 2016 ACM. ISBN 978-1-4503-4019-9/16/01…$15.00
DOI: http://dx.doi.org/10.1145/2866614.2866619

[1]**Card**inalit**y**-based Feature Model **G**ap Detection and

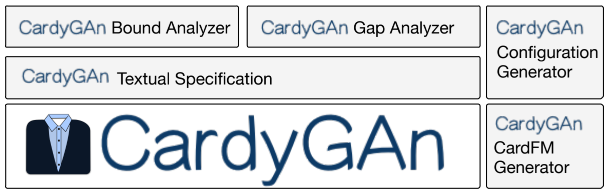

CardyGAn Bound Analyzer
CardyGAn Gap Analyzer
CardyGAn Configuration Generator
CardyGAn Textual Specification
CardyGAn
CardyGAn CardFM Generator

Figure 1: Architecture of CardyGAn

which offers the following functionality.

- **CFM editor** with textual syntax based on Xtext [12], and an integration of cardinality annotations into the exchange format of FeatureIDE [27].
- **CFM validation** for automated anomaly detection for CFM with potentially infinite configuration spaces based on a combination of ILP and SMT solvers.
- **CFM configuration engine** for staged CFM configuration based on Alloy [14].
- **CFM generator** for generating synthetic CFM based on the BeTTy generator [24]. We enrich feature models generated by Betty with randomly chosen cardinality interval annotations according to user-defined selection characteristics.

Figure 1 gives an overview of the different modules of CardyGAn which will be described in more detail in the following. We use an illustrative example being part of a *smart-traffic vehicle-platooning* scenario from the multi-agent domain to describe background and motivation of CFM and the corresponding functionality provided by our tool to support SPL engineering with CFM. The tool is based on our formal specification of CFM syntax and configuration semantics and covers a wide range of existing as well as novel anomalies [28]. To the best of our knowledge, this is the first comprehensive coverage of the CFM formalism in its full extent.

2. BACKGROUND AND MOTIVATION

In this section, we first describe the basic concepts and notions of CFM by means of an illustrative running example.

2.1 Illustrative Example: Vehicle Platooning

Our running example, called *vehicle platooning*, constitutes a highly-configurable multi-agent (software) system from the vehicle-to-x domain [26]. In a vehicle-platooning scenario, various vehicles (agents) are autonomously driving, e.g., on a motorway. By dynamically forming coordinated groups (*platoons*), *platoon members* may reduce headway compared to individual driving, thus potentially leading to increased traffic efficiency, decreased fuel consumption etc. Each platoon member may take one of two possible *roles*. (1) Exactly one member per platoon is nominated as *Leader* who coordinates, e.g., joining and leaving requests of members, whereas (2) an arbitrary, a-priori unrestricted number of other members are called *Followers*.

Analysis Tool. Our tool and additional documentation may be downloaded at https://github.com/Echtzeitsysteme/cardygan

Platoon members use *Communication Channels* to exchange status information and to notify other members, e.g., about speed changes (either *Fast Moving* or *Slow Moving*). Depending on various factors such as the speed of the platoon, corresponding constraints on the *selection/deselection* of communication-channel configuration options are imposed (e.g., *Reliability*, *High Throughput*, and *Addressing Scheme*). Those constraints not only affect the selection, or deselection of communication-channel configuration options, but also the *amount* of available resources, e.g., the number of communication channels available. As a consequence, the software of each individual platoon member is highly (re-)configurable to continuously *adapt* itself to the behaviors of other platoon members as well as ever-changing environmental contexts as apparent in realistic traffic situations.

To this end, *Dynamic Software Product Lines* (DSPL) constitute a comprehensive methodology to specify variable, run-time reconfigurable/adaptable software systems [13]. In particular, we use *feature models* graphically represented as FODA *feature diagrams* [15], to specify (re-)configuration options of our running example within the problem space. In FODA feature diagrams, configuration options are represented as (Boolean) features, each either being selected or deselected in a configuration. However, as illustrated by our running example, not only the presence and absence of feature *types*, but also the *amount* of instances of those feature types may be a crucial configuration decision. This makes CFM an appropriate formalism for variability modeling of systems with an explicitly configurable amount of resources.

2.2 Graphical Syntax and Configuration Semantics of CFM

The CFM for the configuration of one particular *Platoon* is shown in Fig. 2. Similar to the FODA notation, CFM represent the set of configuration options (*features*) (depicted as rectangles), in a hierarchical tree-like diagram, representing a *decomposition* relation among the features [15]. All direct sub-features of one particular feature node form a *feature group* which is visualized with an arc connecting the feature edges of the group members.

As a key concept of CFM, features are not only selected or deselected in a configuration, but rather for each feature *type*, multiple feature *instances* may occur together with copies of their corresponding sub-trees within CFM configurations [9]. Thus, CFM provides UML-like feature multiplicities to specify the allowed number of feature instances being selectable for a particular feature type [22]. In particular, *cardinality intervals* of the form $\langle l, u\rangle$ and $[l, u]$ specify valid selections of feature types and feature instances where a natural number l (including 0) denotes the *lower* bound and a natural number u denotes the *upper* bound for the allowed number of feature types/instances [22]. In addition, the special symbol $*$ is used to represent unrestricted upper bounds.

More specifically, the CFM language comprises the following constructs.

- *Feature instance cardinality*, denoted as $\langle l, u\rangle$ on the left-most position on top of each feature rectangle, restricts the minimum and maximum number of feature instances selectable from the sub-tree clone of respective parent feature instances. As a convention, the root feature (*Platoon* in our example) always has feature instance cardinality $\langle 1, 1\rangle$. Furthermore, $\langle 1, 1\rangle$ de-

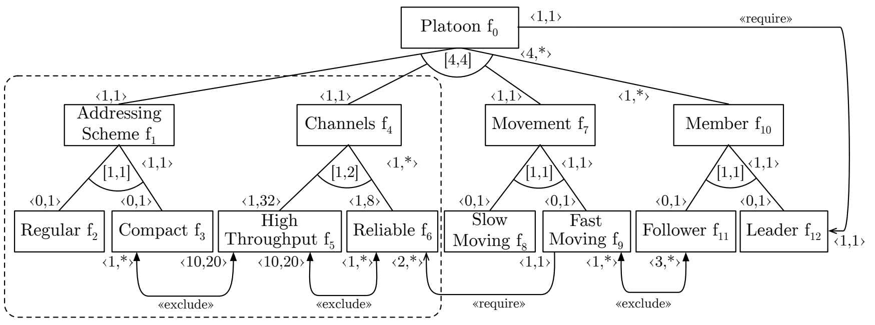


Figure 2: Cardinality-based Feature Model of Vehicle Platooning Running Example

notes that exactly one *Addressing Scheme* is selectable per *Platoon*, whereas $\langle 1, * \rangle$ denotes that each *Platoon* consists of at least one *Member*, i.e., a platoon may comprise an arbitrary number of *Member*s.

- *Feature group type cardinality*, denoted as $[l, u]$, restricts the minimum and maximum number of feature types selectable from the group of direct sub-feature instances. In our example, $[1, 1]$ denotes that in each group of a *Member* instance, either a *Leader* or a *Follower* must be present. In contrast, $[1, *]$ denotes that at least one sub-feature type per *Channel* instance must be selected.

- *Feature group instance cardinality*, denoted as $\langle l, u \rangle$ at the right-hand side of each group arc, restricts the minimum and maximum number of feature instances selectable from the group of direct sub-feature instances. For example, $\langle 1, * \rangle$ denotes that arbitrary many, but at least one instance of *High Throughput* and *Reliable* are selectable for each *Channel* instance.

- *Cross-tree edges*, which may be either *require edges* (unidirectional arrow) or *exclude edges* (bi-directional arrow), are annotated with $\langle l, u \rangle$ cardinality intervals at the source and target features rectangles, and impose constraints on the number of instances of hierarchically unrelated features. In our example, the require edge from *Fast Moving* with source interval cardinality $\langle 1, 1 \rangle$ to *Reliable* with target interval cardinality $\langle 2, * \rangle$ denotes that if one instance of *Fast Moving* is selected, then two or more instances of *Reliable* must be selected. The exclude edge from *Follower* with interval cardinality $\langle 3, * \rangle$ to *Fast Moving* with interval cardinality $\langle 1, * \rangle$ denotes that if there are more than three instances of *Follower*, then no instance of *Fast Moving* may be selected and vice versa.

The *configuration* semantics of FODA feature models consists of Boolean configuration options denoting for each feature if it is selected or not. In contrast, a CFM configuration consists of multiple instances per feature, where each instance is part of a sub-tree which results from cloning the corresponding sub-tree of the parent feature. Figure 3 shows sample configurations C_1, C_2 and C_3 of the CFM in Fig. 2. Each rectangle represents a feature instance labeled with the name of the respective feature type. Each feature instance constitutes the root of a (recursively) cloned sub-tree which may be configured individually for that particular instance.

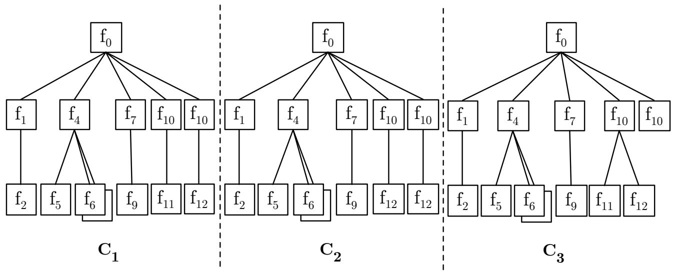


Figure 3: Example Configurations for Running Example

For example, in C_1, a (cloned) feature instances of *Member* may be individually configured within a cloned sub-feature instance of either *Follower* or *Leader*.

Concerning cardinality intervals annotated to cross-tree edges, two kinds of semantics are described in the literature, referring to *local* and *global* interpretation [17]. In this regard, in a global interpretation cardinality interval constraints are imposed on the entire set of feature instance of a CFM, whereas in a local interpretation, interval constraints are imposed for each sub-tree clone individually. In particular, configuration C_1 is valid in case of a global interpretation as exactly one instance of *Leader* is selected. In contrast, C_1 is invalid in case of a local interpretation as C_1 contains a sub-tree clone with no instance of *Leader*. Accordingly, C_2 is invalid in case of a global interpretation as two instances of *Leader* exist, whereas C_2 is valid in case of a local interpretation as each cloned sub-tree of *Member* contains an instance of *Leader*. In the following, we apply the global interpretation in our CFM semantics.

2.3 Inconsistency and Anomalies of CFM

Feature models, although being syntactically well-formed, may expose undesirable *semantic* properties, which are usually subsumed under the notion of *anomalies* [3]. Examples for anomalies in FODA feature diagrams are, e.g., false optional features and dead features [24]. First proposals exist to lift this anomaly notion also to CFM, e.g., by means of dead cardinality anomalies [18]. However, other approaches do neither take into account all possible combinations of cardinality annotations and respective constraints, nor do they consider anomalies in potentially unbounded cardinality intervals. Hence, in addition to dead cardinality anomalies, we define further novel CFM anomalies, e.g., false unbounded intervals. A CFM is *false unbounded* if at least one $*$ occurs in some cardinality interval, but its constraints only allows a finite number of configurations. For instance, a feature instance cardinality interval is *false unbounded*, if its upper bound is specified as $*$, whereas an actual finite upper bound is never exceeded in any valid configuration. As a result, a CFM is *false unbounded* if all potentially unbounded intervals are *false unbounded*.

The excerpt marked by the dashed box in Fig. 2 illustrates several subtle cases of CFM anomalies. For example, the group instance cardinality $\langle 1, * \rangle$ of feature CHANNELS is *false unbounded* as the maximum number of child feature instances is 40. The same holds for cross-tree edge intervals $\langle 1, * \rangle$ relating COMPACT and RELIABLE as the corresponding feature instance intervals are bounded. In contrast, the group instance interval of PLATOON and the feature instance interval of MEMBER are indeed unbounded, thus making the entire CFM unbounded. Besides (false) unbounded anomalies, the example contains further anomalies concerning lower bounds of cardinality intervals. For example, the group type interval $\langle 1, 2 \rangle$ of CHANNELS has an actual lower bound of 2, as both child features must be selected. The same holds for the group instance interval $\langle 1, * \rangle$ of CHANNELS, where the actual lower bound is 2. Besides CFM anomalies affecting upper and/or lower bounds of cardinality intervals, dead cardinalities might be further located within intervals, thus imposing *interval gaps*. For example, the feature instance interval of HIGH THROUGHPUT exposes an interval gap in the sub-range $(10, 20)$, as there exists no valid CFM configuration having a number of feature instances within this sub-range. As a consequence, the group instance interval of CHANNELS exposes an interval gap in sub-range $(18, 21)$ as within this interval, no valid combination of HIGH THROUGHPUT and RELIABLE exists.

For validation of CFM, i.e., for interval-bound analysis and gap detection, we employ formal configuration semantics based on multi-sets. A valid configuration, therefore, contains for each feature type the number of selected feature instances. For bound analysis and gap detection of CFM, a multi-set representation constitutes a reasonable encoding as a valid multi-set configuration includes *at least one* valid cloned sub-tree representation. For example, in Fig. 3 the configurations C_1 and C_3 correspond to the same configuration if represented as multi-sets, as the number of feature instances for each feature type is the same in both configurations. In contrast, for validating concrete CFM configurations, a multi-set representation is insufficient. For instance, configuration C_3 is invalid, as one (cloned) sub-tree of instance f_{10} contains no feature instance, while the other sub-tree includes two instances.

3. TOOL SUPPORT FOR CFM

Variability models such as feature models constitute a key formalism for modeling problem space variability throughout the entire *software product line engineering* process, i.e., domain engineering as well as application engineering [7, 2]. During *domain engineering*, feature models are used for *domain analysis* to capture all relevant domain features together with respective configuration constraints within the problem domain under consideration. Hence, tool support for domain engineering comprises comprehensive feature-model editing [27], as well as automated configuration-constraint analysis capabilities, e.g., for inconsistency checking and anomaly detection [3]. During *application engineering*, feature models are used to repetitively derive valid product configurations, e.g., based on a well-defined *staged configuration process* [8]. As a consequence, respective tool support for application engineering comprises a configuration engine to assist customers during (semi-)automated derivation of product configurations.

However, although rich tool support for both domain and application engineering specifically tailored to FODA feature diagrams already exists, only few attempts can be found so far that support editing, automated analysis, and/or staged configuration of CFM [1, 21, 25]. In the following, we present CARDYGAN, to provide comprehensive tool support for CFM covering *domain engineering* and *application engineering*.

3.1 CFM Editing

As illustrated by the example in Sect. 2, the constructs of CFM and their complicated interplay even in case of small- and medium-sized models require editing assistance during creation and editing of CFM. To this end, CARDYGAN includes a CFM editor which provides a compact and concise textual syntax implemented in the well-established ECLIPSE-based XTEXT framework [12]. In this way, CFM instances can be seamlessly handled by off-the-shelf source-management tools. Figure 4 shows the textual representation of our running example from Fig.2.

- Features are denoted by the keyword `Feature`, preceded by cardinality-interval specifications of the respective feature instances.
- Feature groups are annotated with `@type` to denote group-type cardinality intervals.
- Feature groups are annotated with `@instance` to denote group-instance cardinality intervals.
- The tree-like hierarchy is represented by nested feature/group entities.
- Cross-tree edges are specified separately by referring to the respective source and target features, annotated with corresponding cardinality intervals.

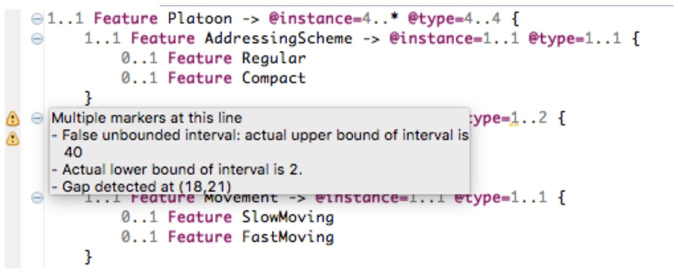
In addition to syntax highlighting and auto-completion provided by the XTEXT-framework by default, we further enrich the CFM editor with capabilities to display results of validity checks as illustrated in Fig. 5.

CARDYGAN supports the *FeatureIDE* exchange format in terms of import/export capabilities for CFM instances. In particular, we employ the commentary fields of features in *FeatureIDE* to specify cardinality interval annotations. CARDYGAN parses those comments according to our textual syntax. If no intervals are defined, feature models are interpreted as FODA feature models as follows.

- Setting 0 (optional) or 1 (mandatory) as lower bound and 1 as upper bound for feature instance intervals.
- For XOR- and OR-Groups, the feature group type intervals $(1, 1)$ and $(1, u)$ are used, where u is the number of features in a group, and similarly for feature group instance intervals.
- For cross-tree edges, we set $(1, 1)$ for both source and target cardinality intervals.

```
1..1 Feature Platoon -> @instance=4..* @type
    =4..4 {
  1..1 Feature AddressingScheme -> @instance
      =1..1 @type=1..1 {
    0..1 Feature Regular
    0..1 Feature Compact
  }
  1..* Feature Channels -> @instance=1..* @type
      =1..2 {
    1..8 Feature HighThroughput
    1..32 Feature Reliable
  }
  1..1 Feature Movement -> @instance=1..1 @type
      =1..1 {
    0..1 Feature SlowMoving
    0..1 Feature FastMoving
  }
  1..* Feature Node -> @instance=1..1 @type
      =1..2 {
    0..1 Feature Follower
    0..1 Feature Leader
  }
}
1..* Compact exclude 10..20 HighThroughput
1..* HighThroughput exclude 3..* Reliable
1..1 FastMoving require 2..* Reliable
3..* Follower exclude 1..* FastMoving
1..1 Platoon require 1..1 Leader
```

Figure 4: Textual Representation of Running Example CFM



```
1..1 Feature Platoon -> @instance=4..* @type=4..4 {
    1..1 Feature AddressingScheme -> @instance=1..1 @type=1..1 {
        0..1 Feature Regular
        0..1 Feature Compact
    }
                                              :ype=1..2 {

    1..1 Feature Movement -> @instance=1..1 @type=1..1 {
        0..1 Feature SlowMoving
        0..1 Feature FastMoving
    }
```

```
Multiple markers at this line
- False unbounded interval: actual upper bound of interval is
  40
- Actual lower bound of interval is 2.
- Gap detected at (18,21)
```

Figure 5: CFM Editor Displaying Results of Validity Checks

3.2 CFM Validation and Anomaly Detection

In our CFM editor, syntactical well-formedness is enforced by the textual syntax. Thereupon, CardyGAn incorporates capabilities for validation and anomaly detection of CFM during domain analysis. Due to the predominant role of cardinality constraints in CFM, any kind of potential semantic inconsistency can be explained through the notion of dead cardinality. As a consequence, we observe two potential causes for anomalies in CFM due to faulty declarations of cardinality intervals: (1) unsatisfiable lower/upper bounds (including false unbounded), and (2) unsatisfiable sub-ranges (gaps). For (1), CardyGAn provides a *Bound Analyzer*, where CFM semantics is encoded as ILP (Integer Linear Programming) representation and we use a respective ILP-solver for bound analysis. For (2), CardyGAn provides a *Gap Analyzer*, where we apply an SMT-solver to detect interval gaps.

Bound Analyzer. An ILP consists of a set of linear inequalities on a set of k integer-valued decision variables. The resulting convex hull forms the feasible region within a k-dimensional search space. An objective function states that either lower (minimum), or upper (maximum) boundary integer values for decision variables should be found by an ILP-solver. For ILP encoding, we employ a multi-set representation for CFM. As a result, if a CFM is reported satisfiable by the ILP-solver, there exists at least one concrete CFM configuration representable in terms of a cloned sub-tree hierarchy. In addition, determining lower and upper bounds based on the multi-set representation, therefore, yields actual lower and upper bounds of all possible CFM configurations. Encoding CFM semantics as ILP thus allows for automated detection of dead cardinality anomalies potentially located at the boundary of cardinality intervals without explicitly enumerating every CFM configuration.

The ILP encoding of the CFM from Fig. 2 is given in Fig. 6. As decision variables, we introduce $\mathfrak{f}_i \in \mathbb{N}_0$, denoting the number of instances of feature f_i, and a *feature selection variable* $\mathfrak{t}_i \in \{0, 1\}$ denoting whether at least one instance of that feature is selected. Consistency between variables $\mathfrak{f}_i$ and $\mathfrak{t}_i$ is enforced by the inequalities in $\boxed{2}$ in Fig. 6. Here, we incorporate a coefficient M, usually referred to as *big M*, by means of a sufficiently large number for coupling binary variables $\mathfrak{t}_i$ to integer variables $\mathfrak{f}_i$. Coefficient M is conservatively approximated by first multiplying maximum upper bound value of cardinality intervals occurring in each branch of the feature tree, and then choosing the overall maximum value. The upper bound is derived from the syntactic context of the cardinality interval under consideration. Occurrences of $*$ are replaced in the same way. The monotonicity of aggregated cardinality interval bound values as imposed by the CFM tree structure, yields correct analysis results when being restricted to M also for unbounded CFM.

To encode CFM semantics of feature instance cardinality intervals and sub-tree cloning for our running example (cf. dashed box in Fig. 2), we introduce inequalities depicted in $\boxed{3}$. For example, the inequality $\mathfrak{f}_4 \leq \mathfrak{f}_5 \leq 32\mathfrak{f}_4$ represents constraints imposed by the feature instance cardinality interval of feature f_5. From each sub-tree, 1 to 32 instances of f_5 are selectable. Hence, the interval bounds are multiplied by $\mathfrak{f}_4$. The inequality restricting the upper bound is only added if the interval is bounded, i.e., the upper bound is not $*$, which does hold in this case. To encode semantics of group instance cardinality intervals, we introduce inequalities depicted in $\boxed{4}$. For example, the inequality $2\mathfrak{f}_0 \leq \mathfrak{f}_1 + \mathfrak{f}_2$ represents the group instance interval of f_0, thus limiting the sum of children instances. Again, for the unbounded case, we only restrict the lower bound. Semantics of group type cardinality intervals are encoded, accordingly (cf. $\boxed{5}$).

Finally, to handle cross-tree edges, we introduce further decision variables $\mathfrak{r}_k \in \{0, 1\}$ denoting a particular interval to be selected or not. For each cross-tree edge, we define inequalities on source and target feature node intervals. For the source feature node f_i, we introduce three interval selection variables $\mathfrak{r}_{k-1}$, $\mathfrak{r}_k$, and $\mathfrak{r}_{k+1}$ to encode selection conditions of the cardinality interval at the source feature of the a cross-tree edge. In our example, the inequalities encoding matching conditions of exclude edges are depicted in $\boxed{6}$ and $\boxed{7}$. For example in $\boxed{6.2}$, inequality $\mathfrak{f}_5 \leq 9\mathfrak{r}_1 + 20\mathfrak{r}_2 + M\mathfrak{r}_3$ encode for interval $(10, 20)$ the lower bounds of matching conditions of interval selection variables. Accordingly, upper

$$\boxed{1}\ \max \mathfrak{f}_1 + \mathfrak{f}_4$$

subject to:

$$\boxed{2}\ \begin{aligned} &\mathfrak{t}_0 \leq \mathfrak{f}_0 \leq M\mathfrak{t}_0 \\ &\mathfrak{t}_1 \leq \mathfrak{f}_1 \leq M\mathfrak{t}_1 \\ &\ldots \end{aligned}$$

$$\boxed{3}\ \begin{aligned} &\mathfrak{f}_0 = 1 \\ &\mathfrak{f}_0 \leq \mathfrak{f}_1 \leq \mathfrak{f}_0 \\ &0 \leq \mathfrak{f}_2 \leq \mathfrak{f}_1 \\ &0 \leq \mathfrak{f}_3 \leq \mathfrak{f}_1 \\ &\mathfrak{f}_0 \leq \mathfrak{f}_4 \leq \mathfrak{f}_0 \\ &\mathfrak{f}_4 \leq \mathfrak{f}_5 \leq 32\mathfrak{f}_4 \\ &\mathfrak{f}_4 \leq \mathfrak{f}_6 \leq 8\mathfrak{f}_4 \end{aligned}$$

$$\boxed{4}\ \begin{aligned} &2\mathfrak{f}_0 \leq \mathfrak{f}_1 + \mathfrak{f}_4 \\ &\mathfrak{f}_1 \leq \mathfrak{f}_2 + \mathfrak{f}_3 \leq \mathfrak{f}_1 \\ &\mathfrak{f}_4 \leq \mathfrak{f}_5 + \mathfrak{f}_6 \end{aligned}$$

$$\boxed{5}\ \begin{aligned} &\mathfrak{t}_0 \leq \mathfrak{t}_1 + \mathfrak{t}_4 \leq 2\mathfrak{f}_0 \\ &\mathfrak{t}_1 \leq \mathfrak{t}_2 + \mathfrak{t}_3 \leq 2\mathfrak{f}_1 \\ &\mathfrak{t}_4 \leq \mathfrak{t}_5 + \mathfrak{t}_6 \leq 2\mathfrak{f}_4 \end{aligned}$$

$$\boxed{6.1}\ \mathfrak{t}_3 = \mathfrak{r}_0$$

$$\boxed{6.2}\ \begin{aligned} &\mathfrak{f}_5 \leq 9\mathfrak{r}_1 + 20\mathfrak{r}_2 + M\mathfrak{r}_3 \\ &\mathfrak{f}_5 \geq \mathfrak{t}_5 - \mathfrak{r}_1 + 9\mathfrak{r}_2 + 20\mathfrak{r}_4 \\ &\mathfrak{t}_5 = \mathfrak{r}_1 + \mathfrak{r}_2 + \mathfrak{r}_3 \end{aligned}$$

$$\boxed{6.3}\ \mathfrak{r}_0 + \mathfrak{r}_2 \leq 1$$

$$\boxed{7.1}\ \mathfrak{t}_5 = \mathfrak{r}_4$$

$$\boxed{7.2}\ \begin{aligned} &\mathfrak{f}_6 \leq 2\mathfrak{r}_5 + M\mathfrak{r}_6 \\ &\mathfrak{f}_6 \geq \mathfrak{t}_6 - \mathfrak{r}_5 + 2\mathfrak{r}_6 \\ &\mathfrak{t}_6 = \mathfrak{r}_5 + \mathfrak{r}_6 \end{aligned}$$

$$\boxed{7.3}\ \mathfrak{r}_4 + \mathfrak{r}_6 \leq 1$$

Figure 6: ILP Encoding of Running Example.

```
Platoon$i1 {
  AddressingScheme$i2
  Channels$i3 {
    Reliable$4
    Reliable$5
    HighThrougput$6
  }
  Movement$i7 {
    FastMoving$i8
  }
  Member$i9 {
    Leader$i10
  }
  Member$i11 {
    Follower$i12
  }
}
```

Figure 7: CardyGAn's Configuration Language Supporting a Staged Configuration Process

bounds are encoded by inequalities $\mathfrak{f}_5 \geq \mathfrak{t}_5 - \mathfrak{r}_1 + 9\mathfrak{r}_2 + 20\mathfrak{r}_4$. In addition, the constraint $\mathfrak{t}_5 = \mathfrak{r}_1 + \mathfrak{r}_2 + \mathfrak{r}_3$ ensures intervals not to be selected and deselected at the same time if f_5 is present. In this case, when the matching condition holds, $\mathfrak{r}_2$ is set to value 1. Due to symmetry of exclude-edges, target feature node cardinality intervals are encoded accordingly (see $\boxed{6.1}$ and $\boxed{7.1}$). Compared to $\boxed{6.2}$ and $\boxed{7.2}$, the set of inequalities is reduced to a single equation due to cross-tree edge cardinality interval $(1, *)$ at f_3 and f_6. To ensure mutual exclusion, inequalities such as $\boxed{6.3}$ and $\boxed{7.3}$ are added.

Based on the ILP encoding, CFM bound analysis is applied by an ILP objective function either stating minimization for lower bound analysis, or maximization for upper bound analysis. In Fig. 6, we analyze the upper bound of the group instance cardinality interval of f_0 by using the objective function max $\mathfrak{f}_5 + \mathfrak{f}_6$ (cf. $\boxed{1}$), which yields 40. For unbounded cardinality intervals, we consider two cases. For false unbounded intervals, e.g., the upper bound of the group instance cardinality of f_4, the solver returns a bounded result with objective value less than M. For true unbounded cardinality intervals, e.g., the upper bound of feature instance cardinality of f_{10}, the solver either reports unbounded but feasible, or it returns value M. As ILP-solvers we support *CPlex*[2], *GLPK*[3] and *Gurobi*[4]. For analysis of FODA feature models, either ILP-solvers or a SAT-solver can be used.

Gap Analyzer. The ILP approach is not applicable for interval-gap detection as gaps are, by definition, not located at minima/maxima locations of the search space. Hence, finding interval gaps is not an optimization problem, but rather a constraint satisfaction problem with integer inequalities. Hence, we apply an SMT-solver equipped with linear integer arithmetics theory according to our ILP encoding of CFM semantics (cf. Fig. 6). Since gap analysis using SMT-solvers is presumably more time consuming than a ILP-based bound analysis, *Gap Analyzer* runs SMT-solver calls in batch mode rather than on-the-fly. CardyGAn currently supports Z3 [11] as SMT-solver.

3.3 CFM Staged Configuration

A staged configuration process of FODA feature models comprises a sequence of configuration steps, i.e., a selection/deselection decisions for every feature, finally ending up in a valid product configuration [8]. However, staged configuration processes of CFM not only consist of sequences of selection/deselection of *types* of features but rather the particular feature *instance* with its corresponding parent feature instance. In this regard, CardyGAn provides support for (staged) configuration of CFM by means of (1) a textual configuration language and a corresponding *CFM configuration editor*, and (2) a configuration validator for staged configuration processes based on Alloy. In addition, we provide a *CFM configuration generator* for automated generation of sample CFM configurations.

Configuration Language. The textual representation of the CFM configurations (cf. Fig. 3) for our running example CFM (cf. Fig. 2) is depicted in Fig. 7. Each derived feature instance is identified by its feature type, followed by an arbitrarily chosen instance identifier. Similar to the textual CFM syntax, the tree structure is specified by nested instance entities denoted by curly brackets. This way, an explicit assignment of feature instances to their corresponding subtree clones is ensured by the configuration language. For example, `AddressingScheme$i1` refers to the first instance of feature type *AddressingScheme*, nested in the parent feature instance `Platoon$i1` of feature type *Platoon*. Apart from fully specified CFM configurations as shown in Fig. 7, the configuration editor also supports valid *partial* CFM configurations being completable to valid full configuration as occurring during a staged configuration process.

Configuration Validator. A CFM configuration specified in the textual syntax shown in Fig. 7 may be either valid or invalid for a given CFM. For automatically validating a (partial) configuration CardyGAn includes a configuration validator implemented in Alloy [14]. Alloy is a language for specifying computational structures defined by a set of relational constraints.

We give a brief description of our representation of CFM configuration semantics in Alloy using the sample configuration from Fig. 7. The corresponding signatures (cf. Fig. 8)

[2]IBM CPlex Solver, http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

[3]GLPK, http://www.gnu.org/software/glpk/glpk.html

[4]Gurobi Solver, http://www.gurobi.com

are defined to represent key entities of CFM syntax as well as CFM configurations as shown in Fig. 7. For example, the compound signature `Instance` encapsulates a relation *feature*, referring to the signature `Feature` corresponding a feature type, as well as a relation `parentInstance` referring to at most one (declared by the keyword `lone`) parent instance. As a result, feature instances within configurations are uniquely identified by their position within a (cloned) sub-tree. In this way, an Alloy encoding of the model entities occurring in a CFM and a corresponding configuration is derived by inheriting from the respective signatures. Thereupon, the formalization of CFM configuration semantics in ALLOY is defined by facts and predicates. For instance, the fact in Fig. 9 imposes the CFM feature hierarchy as well as all derivable CFM configurations to form rooted trees.[5] Invoking the *Alloy Analyzer* [14] with a CFM and a *full* CFM configuration either returns this particular instance in case of a valid configuration or it reports unsatisfiability if it is invalid. In case of a valid *partial* CFM configuration, the invocation yields a valid ALLOY model instance, i.e., a full configuration arbitrarily completing the partial configuration. In case of invalid partial configurations, no valid model instance is found. In this manner, the user can incrementally configure, validate and refine CFM configurations during a staged configuration process.

Finally, CARDYGAN can be used to randomly generate valid configurations of given CFM specifications. This functionality can be used as a starting point for the staged configuration process.

4. RELATED WORK

Related Tools. FEATUREIDE provides a graphical editor as well as support for staged configuration for FODA feature models, but without cardinality annotations. Additionally, FeatureIDE provides capabilities for mapping onto solution-space artifacts which is not considered in CARDYGAN. Until now, only few tools for specification and analysis of CFM exist. *Clafer* integrates class models and cardinality-based feature models in one language and, therefore, also supports specification of CFM [1]. *Clafer* further allows to generate concrete instances, which is similar to our configuration generator in CARDYGAN. Also similar to CARDYGAN, the *Clafer configurator* supports a staged configuration process as well as model validation and error highlighting. However, *Clafer* provides no similiar mean for interval analysis as our tool does. *SALOON* also supports CFM editing, extended with attributes and corresponding constraints, as well as CSP-based CFM analysis capabilities. In contrast, CARDYGAN supports potentially unbounded configuration spaces, as well as staged configuration capabilities.

Analysis of CFM. Riebisch et al. were the first to propose UML-like multiplicities for features in FODA feature models [22]. Thereupon, Czarnecki et al. in [8] further extend feature models with group type and feature instance cardinality, but forbid combinations of both. Czarnecki et al. propose to define semantics of CFM based on sub-tree clones and present a translation into context-free grammar [10, 9]. They also allow unbounded cardinality intervals but do not further investigate their semantic impact. Quinton et al. provide source and target cardinality for require-edges [18]. Nevertheless, their approach does neither consider exclude-constraints, nor the combination of feature instance and group cardinality intervals. Quinton et al. also present a definition of inconsistent CFM similar to our concept of dead cardinality anomaly and detect inconsistencies using a CSP-based encoding [18, 20]. In contrast, Cordy et al. as well as Zhang et al. analyze inconsistencies of CFM using BDD [29, 6]. However, all of these works neither address unboundedness nor interval gaps in their CFM semantics and their analysis. Michel et al. in [17] investigate semantic ambiguities caused by combinations of feature and group cardinality and distinguish local clone-based from global feature-based interpretation of group type cardinality intervals, which is similar to our notion of group instance and group type cardinality intervals. However, they only consider a global feature-based interpretation, being similar to our notion of group type cardinality intervals. Cordy et al. allow the combination of feature and group cardinality intervals, but limit the latter to our notion of group type cardinality intervals [6] . Again, neither Michel et al. nor Cordy et al. consider unboundedness semantically or during CFM analysis.

5. CONCLUSION

In this paper, we presented our CFM tool CARDYGAN for supporting modeling and editing, analyzing, and configuring of cardinality-based feature models. As future work, we plan to evaluate the different components of our tool by means of synthetically generated experimental data as well as real-world case studies such as our running example. In addition, we plan to define mapping concepts of CFM onto solution space artifacts in order to establish a fully-fledged SPL engineering methodology incorporating CFM.

6. ACKNOWLEDGMENTS

This work has been funded by the German Research Foundation (DFG) as part of projects A01 and B01 within the Collaborative Research Center (CRC) 1053 – MAKI. This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP 1593: Design For Future – Managed Software Evolution.

7. REFERENCES

[1] M. Antkiewicz, K. Bak, A. Murashkin, R. Olaechea, J. H. J. Liang, and K. Czarnecki. Clafer Tools for Product Line Engineering. In *SPLC 2013 workshops*, pages 130–135, 2013.

[2] S. Apel, D. S. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 1. edition, 2013.

[3] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years later: A Literature Review. *Information Systems*, 35, 2010.

[4] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated Reasoning on Feature Models. In *CAiSE*, pages 491–503, 2005.

[5] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing, 1. edition, 2001.

[6] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Beyond Boolean Product-line Model

[5]The complete ALLOY specification for abstract CFM syntax and configuration semantics for our running example may be found at https://github.com/Echtzeitsysteme/cardygan/tree/master/spec/alloy

```
one sig FM {
 root: one Feature,
 group: set Feature,
 require: ConstraintNode -> ConstraintNode,
 exclude: ConstraintNode -> ConstraintNode,
 config: set Instance
}
abstract sig Feature{
 groupCardinality: set Interval,
 groupInstanceCardinality: set Interval,
 cardinality: some Interval,
 parent: lone Feature,
 instances: set Instance
}
```

```
sig Instance {
 feature: one Feature,
 instanceParent: lone Instance
}
sig Interval {
 lowerBound: Int,
 upperBound: Int + StarSymbol
}
abstract sig ConstraintNode {
 feature: one Feature,
 cardinality: some Interval
}
one sig StarSymbol {}
```

Figure 8: Meta-Model for Cloned Sub-tree Configurations in Alloy

```
fact rootedTree {
  all f: Feature | FM.root in f.*parent
  all f: FM.root | no f.parent
  all f: Feature - FM.root | one f.parent
  no f: Feature | f in f.^parent
}
```

Figure 9: Sample Fact for CFM Configurations

Checking: Dealing with Feature Attributes and Multi-features. In *Proc. of ICSE'13*, pages 472–481, 2013.

[7] K. Czarnecki and U. W. Eisenecker. *Generative programming - methods, tools and applications.* Addison-Wesley, 1. edition, 2000.

[8] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration using Feature Models. In *Software Product Lines*, pages 266–283. 2004.

[9] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[10] K. Czarnecki and C. H. P. Kim. Cardinality-based Feature Modeling and Constraints: A Progress Report. In *International Workshop on Software Factories at OOPSLA*, 2005.

[11] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proc. of TACAS'08*, pages 337–340, 2008.

[12] M. Eysholdt and H. Behrens. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Proc. of OOPSLA'10*, pages 307–309, 2010.

[13] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. *Computer*, 41(4):93–95, 2008.

[14] D. Jackson. *Software Abstractions.* Revised edition, 2012.

[15] 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.

[16] M. Mendonça, A. Wasowski, and K. Czarnecki. SAT-based Analysis of Feature Models is Easy. In *Proc. of SPLC'09*, pages 231–240, 2009.

[17] R. Michel, A. Classen, A. Hubaux, and Q. Boucher. A Formal Semantics for Feature Cardinalities in Feature Diagrams. In *Proc. of VaMoS'11*, pages 82–89, 2011.

[18] C. Quinton, A. Pleuss, D. L. Berre, L. Duchien, and G. Botterweck. Consistency Checking for the Evolution of Cardinality-based Feature Models. In *Proc. of SPLC'14*, pages 122–131, 2014.

[19] C. Quinton, D. Romero, and L. Duchien. Cardinality-based Feature Models with Constraints: A Pragmatic Approach. In *Proc. of SPLC'13*, pages 162–166, 2013.

[20] C. Quinton, D. Romero, and L. Duchien. Automated Selection and Configuration of Cloud Environments Using Software Product Lines Principles. In *Proc. of CLOUD'14'*, pages 144–151, June 2014.

[21] C. Quinton, D. Romero, and L. Duchien. SALOON: A Platform for Selecting and Configuring Cloud Environments. *Software – Practice and Experience*, 46, Jan. 2015.

[22] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending Feature Diagrams with UML Multiplicities. In *IDPT*, 2002.

[23] J. Schroeter, P. Mucha, M. Muth, K. Jugel, and M. Lochau. Dynamic Configuration Management of Cloud-based Applications. In *Proc. of SPLC'12*, pages 171–178, 2012.

[24] S. Segura, J. Galindo, D. Benavides, J. Parejo, and A. Ruiz-Cortés. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In U. Eisenecker, S. Apel, and S. Gnesi, editors, *Proc. of VaMoS'12*, pages 63–71, 2012.

[25] SINTEF (MOD research group). Cvl 2 tool. http://modelbased.net/tools/cvl-2-tool/, 2013.

[26] T. Tank, N. Yee, and J. M. G. Linnartz. Vehicle-to-Vehicle Communication for AVCS platooning. In *Proc. of VTC*, pages 448–451, 1994.

[27] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Sci. Comput. Program.*, 79:70–85, 2014.

[28] M. Weckesser, M. Lochau, T. Schnabel, B. Richerzhagen, and A. Schürr. Mind the Gap! Automated Anomaly Detection for Potentially Unbounded Cardinality-based Feature Models. In *FASE '16*, page accepted to appear, 2016.

[29] W. Zhang, H. Yan, H. Zhao, and Z. Jin. A BDD-Based Approach to Verifying Clone-Enabled Feature Models' Constraints and Customization. In *Proc. of ICSR'08*, pages 186–199, 2008.