2011 35th IEEE Annual Computer Software and Applications Conference

Conformance Checking with Constraint Logic Programming: The Case of Feature Models

Raúl Mazo[1,3], Roberto E. Lopez-Herrejon[2], Camille Salinesi[1], Daniel Diaz[1], Alexander Egyed[2]
[1] CRI, Panthéon Sorbonne University, Paris, France
[2] Institute for Systems Engineering and Automation, Johannes Kepler University, Linz, Austria
[3] Ingeniería de Sistemas, University of Antioquia, Medellín, Colombia
raulmazo@gmail.com, {camille.salinesi, daniel.diaz}@univ-paris1.fr, {roberto.lopez, alexander.egyed}@jku.at

Abstract— **Developing high quality systems depends on developing high quality models. An important facet of model quality is their consistency with respect to their meta-model. We call the verification of this quality the conformance checking process. We are interested in the conformance checking of Product Line Models (PLMs). The problem in the context of product lines is that product models are not created by instantiating a meta-model: they are derived from PLMs. Therefore it is usually at the level of PLMs that conformance checking is applied. On the semantic level, a PLM is defined as the collection of all the product models that can be derived from it. Therefore checking the conformance of the PLM is equivalent to checking the conformance of all the product models. However, we would like to avoid this naïve approach because it is not scalable due to the high number of models. In fact, it is even sometimes infeasible to calculate the number of product models of a PLM. Despite the importance of PLM conformance checking, very few research works have been published and tools do not adequately support it. In this paper, we present an approach that employs Constraint Logic Programming as a technology on which to build a PLM conformance checking solution. The paper demonstrates the approach with feature models, the de facto standard for modeling software product lines. Based on an extensive literature review and an empirical study, we identified a set of 9 conformance checking rules and implemented them on the GNU Prolog constraints solver. We evaluated our approach by applying our rules to 50 feature models of sizes up to 10000 features. The evaluation showed that our approach is effective and scalable to industry size models.**

Keywords-product line models, feature models, conformance checking, verification, constraint logic programming.

I. INTRODUCTION

Information Systems Engineering highly depends on conceptual modeling. As a result, developing high quality systems depends on developing high quality models [23]. Verifying the quality of models has recently been a prominent topic for many researchers in the community. Different kinds of checking have been studied:

- consistency checking [31] consists in "analyzing models to identify unwanted configurations defined by the inconsistency rules";
- model checking [20] consist in verifying "correctness properties of safety-critical reactive systems";
- domain specific properties verification [16,17,18,26] consists in "finding undesirable properties, such as redundant or contradictory information";

In this paper, we are interested in the conformance checking of Product Line Models (PLMs). As many works show it [1,10,24,26], product lines engineering is a specific topic of Systems Engineering that requires adequate models, meta-models, methods, and tools. We are particularly interested in a kind of consistency verification called conformance checking where "it is checked that a model satisfies the constraints captured in the meta-model, i.e., that the model is indeed a valid instance of the meta-model" [32]. The problem in the context of product lines is that verification cannot be achieved at the level of products because these product models are not instantiated from their meta-models, but by configuration of PLMs. The expectation is that conformance checking is achieved at the PLM level, with the assumption that any product model that can be configured from a correct PLM is itself correct. On the semantic level, a product line model is defined as the collection of all the product models that can be derived from it. Therefore checking the conformance of the product line model is equivalent to checking the conformance of all the product models in stage configuration [47]. However, we would like to avoid verifying all the product models because their number can be simply too high [17]. The naïve approach that consists in carrying out product model verification by checking late their conformance with the product line meta-model is also not scalable to real world constraints. We believe that scalable methods, techniques and tools are needed to deal with this important issue [32], which, to the best of our knowledge, is not properly handled by tools. Our literature study revealed that (a) conformance checking approaches that check all the product models of the PLM do not scale to real size models [6], and (b) the checking of larger models is sometimes even unrealizable due to the impossibility to configure all products [16, 17].

To overcome these limitations, we propose an approach to check the conformance of product line models. In this paper, the approach is applied on feature models. The idea of our approach is to test only the elements that are within the scope of each particular conformance rule [6]. The tests are implemented in a declarative way using Constraint Logic Programming (CLP). Conformance rules can be seen as white-boxes allowing special declarations and manipulations (such as the scope-elements in our case). Nine of the rules

0730-3157/11 $26.00 © 2011 IEEE
DOI 10.1109/COMPSAC.2011.66

defined in our approach were evaluated on 50 models of sizes up to 10000 features. The evaluation of each rule demonstrates excellent scalability with performance results being, in any case, less than 140 milliseconds.

Section 2 presents the related work and provides an overview of our approach. Sections 3, 4, and 5 present in a more detailed way how the approach works with feature models. Section 6 presents the implementation and the evaluation of the precision, scalability and usability of our approach. Future works are discussed in section 7.

II. Related Works

There is to our knowledge very few works on the topic of conformance checking, for instance [32] in the UML domain. However, conformance is considered as a kind of consistency [31]. Therefore, this section starts by discussing some of the most well known consistency checking methods that were applied in the PL context, in the light of the conformance checking concern.

Egyed proposed a framework to incrementally detect inconsistencies in UML models [6] and in DOPLER models [25]. This framework first uses inconsistency rules specified with OCL. Each rule starts by identifying the model elements to analyze. Then, all the model elements for which an inconsistency is detected are inserted in a "rule scope" in order to keep track of them. The rule scope consists in a relation between an inconsistency detection rule and the collection of model elements that need to be re-analyzed after they have been corrected. The next time the rule is executed, the check will only be made over the elements in the "rule scope", and not over the complete model. This allows reducing the execution time after the first checking. Egyed presents very efficient performance charts for his approach, but also observes that this approach may not be efficient for all kinds of consistency rules (on a product line domain or others).

In [30] Cabot et al present an object constraint language (OCL) incremental checker. Each time incorrect model elements are identified, a new rule is generated to check that the consistency constraint is satisfied over these specific elements after their correction. However, Blanc et al observe that "OCL description language has a limited usage as it can only describe mono-contextual inconsistencies; in the context of software architecture models it is advocated to target multi-context/multi-paradigm inconsistencies" [31]. This issue is important in the context of conformance checking, as shown in rule 4 (presented in section 5), which is an example of a multi-context conformance rule.

Blanc et al. also propose an incremental inconsistency checker for UML models. Their approach is to use declarative programming-based rules that "analyze the modifications performed on a model in order to identify a subset of inconsistency rules that need to be re-checked" [31]. The analysis uses an impact matrix to represent dependencies between the operations modifying a class and inconsistency rules. With the information provided by the impact matrix users can decide whether and when to execute the incremental check of impacted rules. The problem in the product line context is that modifying a single part of a model can challenge the consistency of all the other elements in the model. In contrast, the impact matrix is efficient only when a few model elements are impacted.

In the domain of databases, one of the aims of consistency checking is to guarantee data integrity and to detect whether data violate integrity constraints. Even if the works in databases are not focused on conformance checking, the approaches used to solve the integrity problems are similar to ours. For example, Kowalski et al. [34] and Olivé [33] proposed declarative-based approaches to check the integrity of deductive databases. These examples show the omnipresence of inconsistencies during the modeling process, the pertinence of declarative programming as a solution to detect inconsistencies in different domains and encourage us to find solutions in order to deal with inconsistencies in new domains as product line engineering.

One of the most popular tools for automatic analysis of software models is Alloy [32]. Alloy works by transforming the model into an instance of SAT (satisfiability of a boolean formula). The need to represent product line models as Boolean formulae limits the usefulness of this approach. Indeed, other kinds of constraints (e.g., arithmetic or symbolic constraints over integer or real variables) are needed in product line models, and cannot be specified with simple Boolean formulae, as Salinesi et al. show in [12, 45]. This is typically the situation in our case where we have to deal with models that contain arithmetic and symbolic constraints.

In the product line domain, there are some tools that provide consistency checking functions. ToolDAy [27] is a product line management tool that guides activities such as scope definition, domain modeling, documentation, consistency checking, and product derivation. SPLOT [9] is a Web-based reasoning and configuration system for cardinality-based feature models. The system maps feature models into propositional logic formulas and uses boolean-based techniques such as binary decision diagram and SAT solvers to reason on PL models. Unfortunately, none of these tools supports conformance checking.

A tool to check conformance of a model with regards to the corresponding meta-model is the EMF Validation Framework which provides a means to evaluate and ensure the well-formedness of EMF models but its use for product line models has not been assessed [46].

In our previous works [29], we presented a tool for the automatic verification of structural correctness of cardinality-based feature models. This tool implemented verification operations such as the identification of redundant features, inconsistent constraints, cyclic relationships and poorly defined cardinalities. The tool used graph navigation algorithms to evaluate each verification criteria, which was effective, but raised major scalability, language-dependency and extensibility issues. The purpose of this paper is to present a new approach that overcomes these issues.

Our new approach belongs to a family of methods [31] [33] and [34], that use CLP to implement model checking. The principle is that rules are implemented with a mix of

logic programming (namely with Prolog), and constraint programming, which is embedded in the Prolog code. This paper is the first one that applies the CLP approach to check conformance of product line models, namely of feature oriented models.

III. Feature Models in a Nutshell

A *feature* is a prominent or distinctive user-visible aspect, requirement, quality, or characteristic of a software system [19]. A *Feature Model (FM)* defines the valid combinations of features in a software product line, and is depicted as tree-like structure in which nodes represent features, and edges the relationships among them [24]. All the nodes are the children of the root node, which is called *root feature* and identifies the product line.

FMs were first introduced in 1990 as a part of the Feature-Oriented Domain Analysis (FODA) method [19], as a means to represent the commonalities and variabilities of software product lines. Since then, feature modeling has become a de facto standard adopted by the software product line community and several extensions have been proposed to improve and enrich their expressiveness. Two of these extensions are cardinalities [11,28] and attributes [8,13,15]. Although there is no consensus on a notation to define attributes, most proposals agree that an attribute is a variable with a name, a domain and a value (by instance, `Intensity` and `Type` are two attributes of the feature `Vibration` of our running example of Figure 1). Note that the value of attributes is not specified in the product line model. Instead, the value of each attribute is assigned for each particular configuration, (when these attributes are attached to features that belong to the configurations). In this paper, we are interested into these two extensions.

In order to handle the semantic of these formalisms, we reason using the abstract syntax instead of the concrete syntax (what the user sees), as recommended by [10]. The outcome is more simplicity and less error-prone analyses. Indeed "the abstract syntax ignores the visual rendering information that is useless to assign a formal semantics to a diagram, e.g., whether nodes are circles or boxes, whether an operator is represented by a diamond shape or by joining the edges departing from a node, etc" [10]. There are two common ways to provide the abstract syntax information [37]: (1) mathematical notation or (2) meta-model. In this paper, we use the second kind of notation because we believe it is the most adequate to our goal of checking conformance of FMs with respect to their meta-model.

According to their meta-model (formalized in section IV) a FM is a DAG (directed acyclic graph) composed of features as nodes and various kinds of relationships:

- *Mandatory:* Given two features `F1` and `F2`, where `F1` is the father of `F2`, a mandatory relationship between `F1` and `F2` means that if `F1` is selected in a product, then `F2` must be selected too, and vice versa.
- *Optional:* Given two features `F1` and `F2`, where `F1` is the father of `F2`, an optional relationship between `F1` and `F2` means that if `F1` is selected in a product, then `F2` may be selected or not. However, if `F2` is selected then `F1` must also be selected.
- *Requires:* Given two features `F1` and `F2`, a relationship `F1` *requires* `F2` means that if `F1` is selected in a product then `F2` has to be selected as well. Additionally, it means that `F2` can be selected even when `F1` is not selected.
- *Exclusion:* Given two features `F1` and `F2`, a relationship `F1` *excludes* `F2` means that `F1` and `F2` cannot be selected in the same product.
- *Group cardinality:* A group cardinality is an interval denoted `<n..m>`, with `n` as lower bound and `m` as upper bound limiting within a group of features the number of features that can be part of a product. All the features in the group must have the same parent feature, and none can be selected if the parent is not itself selected.

As a running example, we illustrate FMs and our work with the example of the Movement Control System (MCS) of a car [12]. In order to illustrate our approach, we intentionally introduced errors in the original model. As the resulting model presented in Figure 1 shows it, a MCS is composed (within others) of sensors and feedback devices that respectively detect movements and position of the vehicle, and sends specific signals to the driver. These features are denoted as mandatory (depicted with a filled circle) called `Sensor` and `Feedback`, respectively. A `Sensor` can measure vehicle movement in two ways: speed and position. These are identified in the model by means of two optional features (depicted with an empty circle) called `Speed Sensor` and `Position Sensor`, respectively. These two features are related in a group cardinality `<1..3>` where `1` is the lower bound and `3` the upper bound limiting the number features that can be selected in a configuration. Of course, the upper bound (`3`) is incorrect, as there are only two features in the group cardinality. Feedbacks to the driver can be of two types: audio and vibration. These are represented by the mandatory feature `Audio` and the optional feature `Vibration`. The audio feedback consists of a warning sound in a defined `Volume`, and the vibration feedback refers to small mechanical oscillations (with `Intensity` and of a certain `Type`) of the steering wheel. Features `Audio` and `Vibration` are related in a group cardinality `<1..1>` where only one feature can be selected in a configuration. In addition, if `Speed Sensor` is activated, the feedback cannot be by vibration, due to security reasons, thus `Speed Sensor` and `Vibration` are modeled as mutual excluded features. Finally, if the `Vibration` feature is selected, its father, the `Feedback`

feature must be selected as well due to the requires-type relationship between `Vibration` and `Feedback`. Example of Figure 1 will be used in the remains of the paper in order to illustrate our approach. It is because, we induced three errors on this model: one upper bound cardinality (1..3), the repeated name of two attributes (Volume) and one redundant relationship (Vibration requires Feedback).

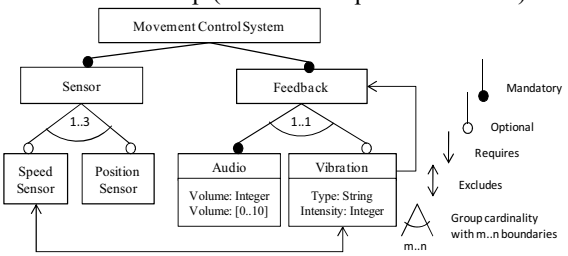
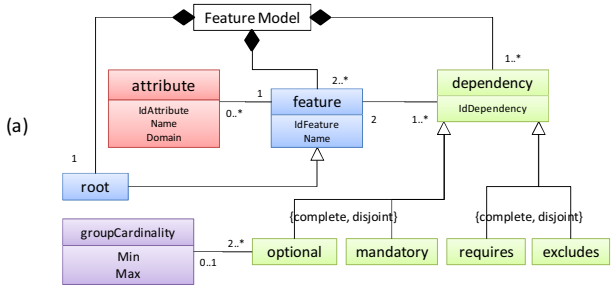


Figure 1. Extract of a car movement control system represented as a feature model.

The FM meta-model used in this paper, see Figure 2, is based on the abstract syntax [37] of SPLOT models [9] augmented with concepts from [10] and [8]. The former adaptation was necessary for allowing attributes in FMs, which are used in our industrial FMs. The FM meta-model is represented as meta-facts as we explain in the rest of this section.



(b)

```
root(IdFeature).
feature( IdFeature, Name, IdAttributes).
attribute( IdAttribute, Name, Domain).
dependency(IdDependency, IdFeature1, IdFeature2).
optional (IdDependency).
mandatory (IdDependency).
requires(IdDependency).
excludes(IdDependency).
groupCardinality(IdDependencies, Min, Max).
```

Figure 2. (a) Feature model meta-model and (b) its representation as meta-facts.

In the meta-model depicted in Figure 2(a), FM's elements are modeled by meta-classes, and relationships between these elements are modeled by meta-associations. In CLP, FM's elements and its relationships are called *meta-fact* and are implemented as CLP facts. In other words, a meta-fact is the CLP structure that represents a fact. In order to define a meta-fact it is necessary to define its name, its parameters and its arity (in case of equal names, the number of parameters make two meta-fact different). The mapping between the FM meta-model as a class diagram in Figure 2(a) and the FM meta-model represented as meta-facts in Figure 2(b) are explained in the rest of this section. Each meta-fact has an attribute that uniquely identifies each instance of the meta-fact. Identifiers are represented as strings (Prolog's atoms) and the references to other FM's entities are represented as lists of identifiers; in both cases, the name of the corresponding variable is preceded by the label `Id`.

```
Meta-fact 1: feature(IdFeature, Name,
IdAttributes).
```

`Name` is a string representing the feature's name and `IdAttributes` is a list of attribute identifiers `[IdAtt1,...,IdAttN]`, where `[]` represents an empty list.

```
Meta-fact 2: root(IdFeature).
```

The root feature (i.e. `Movement Control System`) identifies the product line. In this meta-fact the attribute `IdFeature` references to the root feature.

```
Meta-fact 3: attribute(IdAttribute, Name,
Domain).
```

An attribute has an identifier, a name and a domain. `Name` is a string representing the name of the attribute instantiated with this meta-fact. `Domain` is a collection of values that can take the attribute. For example `['read']` means that the value of the corresponding attribute can be only `'read'`; `[1..5]` means that the value of the corresponding attribute can be an integer between `1` and `5`; `[integer]` means that the value of the corresponding attribute must be an integer.

```
Meta-fact 4: dependency(IdDependency, IdFeature1,
IdFeature2).
Meta-fact 5: optional(IdDependency).
Meta-fact 6: mandatory(IdDependency).
Meta-fact 7: requires(IdDependency).
Meta-fact 8: excludes(IdDependency).
```

Relationships between two features are represented by meta-fact 4. In this meta-fact, `IdFeature1` and `IdFeature2` respectively represent the identifiers of the initial and target features intervening in the dependency. Dependencies can be of four types: `mandatory`, `optional`, `requires`, or `excludes`, respectively represented by meta-facts 5, 6, 7 and 8. Each meta-fact from 5 to 8 references the corresponding dependency. For example, an `optional` dependency references the corresponding dependency having the identifiers of the parent and child features (`IdFeature1` and `IdFeature2` respectively) intervening in the optional dependency. In requires dependencies `IdFeature1` is the requiring feature and `IdFeature2` represents the required feature.

```
Meta-fact 9: groupCardinality(IdDependencies, Min,
Max).
```

Cardinality is a relationship between several features constrained by a `Min` and a `Max` value. Cardinalities can be represented by instantiation of meta-fact 9, where `IdDependencies` is a list of dependency's identifiers related in the group cardinality.

The relationship between the meta-fact and the derived facts respects the basic principle of meta-modeling. In our case, the instantiation of a meta-fact consists in giving constant values to the parameters of this meta-fact. We show this instantiation with our running example. Note that in the following representation of the car MCS as CP facts, each feature, attribute and dependency is identified by a natural number preceded by the label `fea`, `att` and `dep`, respectively.

```
(1) root(fea1).
(2) feature(fea1, 'Movement Control System',[]).
(3) feature(fea2, 'Sensor', []).
(4) feature(fea3, 'Speed Sensor', []).
(5) feature(fea4, 'Position Sensor', []).
(6) feature(fea5, 'Feedback', []).
(7) feature(fea6, 'Audio', [att1,att2]).
(8) feature(fea7, 'Vibration', [att3, att4]).
(9) attribute(att1, 'Volume', [integer]).
(10) attribute(att2, 'Volume', [0..10]).
(11) attribute(att3, 'Type', [string]).
(12) attribute(att4, 'Intensity', [integer]).
(13) dependency(dep1, fea1, fea2).
(14) dependency(dep2, fea1, fea5).
(15) dependency(dep3, fea2, fea3).
(16) dependency(dep4, fea2, fea4).
(17) dependency(dep5, fea5, fea6).
(18) dependency(dep6, fea5, fea7).
(19) dependency(dep7, fea7, fea5).
(20) dependency(dep8, fea3, fea7).
(21) mandatory(dep1).
(22) mandatory(dep2).
(23) optional(dep3).
(24) optional(dep4).
(25) mandatory(dep5).
(26) optional(dep6).
(28) requires(dep7).
(29) excludes(dep8).
(30) groupCardinality([dep3, dep4], 1, 3).
(31) groupCardinality([dep5, dep6], 1, 1).
```

Lines 1 and 2 define root feature `Movement Control System` with no attributes. Lines 3, 4, 5 and 6 define respectively features `Sensor`, `Speed Sensor`, `Position Sensor` and `Feedback` with no attributes. Line 7 defines feature `Audio` with two attributes (`att1` and `att2`) respectively defined in lines 9 and 10. Line 8 defines feature `Vibration` with the attributes `att3` (`Type`) and `att4` (`Intensity`), respectively defined in lines 11 and 12. Lines 13 and 21 define one mandatory dependency between the features `fea1` (`Movement Control System`) and `fea2` (`Sensor`). In the same way, lines 14 to 20 define dependencies between two features of the product line and lines 22 to 29 are facts representing the type of each of these dependencies. Line 30 defines the group cardinality `<1,3>` for dependencies `dep3` and `dep4`. Finally, line 31 defines the group cardinality `<1,1>` for dependencies `dep5` and `dep6`.

IV. Conformance Checking in Feature Models

The conformance of feature model is essential for deriving correct products and enables safe automated reasoning operations such as variability analysis, transformation and code generation [10]. In this way, quality assurance of FMs is essential for successful PL engineering and, due to the ability of FMs to derive a potentially large number of products; any error on the FM will inevitably affect many products of the product line. Besides, the proven benefits of a PL (e.g. reduced time to market, better reuse and therefore reduced development costs and increase in software quality [1,2]) can be compromised by the poor quality of FMs. Therefore, engineers need to be supported in detecting conformance errors during feature modeling.

In this paper we use nine conformance rules that are based on the FM meta-model presented in Figure 2. Our purpose in this paper was not to present an exhaustive list of rules. Rather it was to show how a few relevant rules can be extracted from the meta-model and checked automatically. In this manner, a user of our approach can extend the conformance checking rules according to her/his particular needs. These nine rules were developed based on our experience with verification of product line models of various sizes [38] and the rules found in our literature review. A conformance rule can be seen as a query that will be executed over a FM. If the rule is evaluated `true` in a model, its output is a set of elements that make `true` the evaluation of the rule and by using the backtracking mechanism of CLP solvers we get rest anomaly' sources if any exists. Next we present and formalize our nine rules. Note that in each formalization we (i) specify the scope (elements that need to be analyzed to evaluate this rule) in a general manner, and (ii) specify the case where the conformance rule is evaluated `true`, so, we are not just identifying the presence of an anomaly but also the sources of the anomaly; and (iii) are exhaustive in our search to guarantee completeness of our approach.

Rule 1: A feature should not have two attributes with the same name. In our running example, feature `Audio` is violating this conformance rule because its two attributes have the same name (`Volume`). In the next formula, we are searching two different attributes, of the same feature, with the same name.

Rule 2: Two features should not have the same name. The fact that several features share the same name can imply ambiguity problems in product configuration and maintenance stages.

Rule 3: In our feature-based formalism, product line models should not have more than one root [11, 14, 19, 28]. If a feature model has more than one root feature and our particular feature model formalism allows only one, this rule identifies these root features.

Rule 4: Features intervening in a group cardinality relationship should not be mandatory features. By definition, a cardinality relationship is about the selection of a certain

number of elements among a set of them. In this selection each element must have the same possibility to be chosen than others, that is why elements must be optional features. We consider this rule, presented in [11], as a good practice to avoid errors and redundancies. To illustrate this, let us consider the case of our running example, where `Audio` is a mandatory feature intervening in the `<1..1>` group cardinality. The `<1..1>` means that only one feature can be selected, so, if `Audio` is mandatory, `Vibration` can never be selected.

Rule 5: One feature must not be optional and mandatory at the same time. If a feature is optional, by definition (see section 3.1) it cannot be mandatory and vice versa. In the meta-model, `optional` and `mandatory` are `complete` and `disjoint` dependencies. This rule has two cases. In the first case, rule 5 evaluates if a feature is constrained two times by the same father by means of optional and mandatory relationships. In the second case, rule 5 evaluates if a feature is mandatory towards one parent and optional towards other, directly or indirectly (through other features).

Rule 6: In a group cardinality `<Min..Max>` restricting a set of `N` dependencies (or its associated features), the `Min` and `Max` values must be integers satisfying: $0 \leq Min \leq Max \leq N$. In our running example `SpeedSensor` and `PositionSensor` are participating in the group cardinality `<1..3>`. Note that as only two features are related in the cardinality, the upper value of the cardinality can never be attained. If we apply rule 6 to this particular group cardinality, then $Min = 1$, $Max = 3$ and $N = 2$. According to Czarnecki et al. [11], rule 6 constraints that values of `Min`, `Max` and `N` must be integer numbers and that $0 \leq Min \leq Max \leq N$. But in our running example, we have $Max > N$.

Rule 7: Two features cannot be required and mutually excluded at the same time. If two features are related in requires and excludes relationships, the model is non-conformant. Rule 7 is applicable in the cases in which features are related directly (i.e., F_1 requires F_2 and F_1 excludes F_2) and transitively (i.e., F_1 requires F_2, F_2 requires F_3 and, F_1 excludes F_3) in mutual exclusion and requires.

Rule 8: A root element should not be excluded. If the root feature of a FM is excluded by other feature, the FM becomes void because it does not define any product.

Rule 9: A feature should not require itself or one of its ancestors. In a FM, feature `A` is ancestor of feature `B` if `A` is in the path from the root to `B`. In our running example, `Vibration` is requiring its father `Feedback`, what is a redundancy because `Vibration` can only be selected by the way of `Feedback`.

V. Implementation and Evaluation of the FM Conformance Checking Rules

Our conformance rules are implemented as CLP queries [22, 5], in a way to guarantee termination and exhaustive search [5] using GNU Prolog. Due to space limitations, we do not present the code source of all rules but only of the first one. All the rules are available for download from the tool website[1].

```
(1) conformance_1(FeatureName,AttId1,AttId2,AttName) :-
(2)     feature(_, FeatureName, LAttId),
(3)     chose(LAttId, AttId1, LAttId1),
(4)     member(AttId2, LAttId1),
(5)     AttId1 \== AttId2,
(6)     attribute(AttId1, AttName, _),
(7)     attribute(AttId2, AttName, _).
```

Line 1 uses four output variables to return the name of the feature that has the repeated attributes, their two identifiers and the name of the repeated attributes. These variables will take the values of one feature where two of its attributes have the same name. Usually in Prolog other solutions can be obtained thanks to the underlying non-determinism mechanism. The source of non-determinism are in line 2 that chooses one feature, line 3 that chooses a first attribute of the current feature and line 4, which chooses a second attribute of the current feature. Then line 5 constraints the fact that both features must be different and lines 6 and 7 constraint the fact that the two attributes must have the same name. It is worth noticing the declarative formulation of this conformance check and the fact that we only use relevant elements for the conformance rule (e.g., in this rule we are interested in comparing attributes of a same feature, so, we only consider features with a list of attributes (`LAttId`) and do not use dependencies or cardinalities because they are not relevant for this rule). The research strategy we use to find anomalies with each conformance rule is exhaustive because we do not avoid evaluating any case even if in our research we consider only relevant elements according to the scope of each conformance rule.

We assessed the feasibility, precision and scalability of our approach with 50 models, out of which 48 were taken from the SPLOT repository [9]. The other two models were developed during industry collaboration projects [41,42]. The sizes of the models are distributed as follows: 30 models of sizes from 9 to 49 features, 4 from 50 to 99, 4 from 100 to 999, 9 from 1000 to 9999 and 3 of 10000 features. The domains tackled span from insurance to entertainment, web applications, home automation, search engines, and databases. Note that SPLOT models neither support attributes nor multi root features. Therefore artificial attributes (a variable followed by a domain, for example `A:String`) were introduced in a random way, in order to have models with 30%, 60% or 100% of their features with attributes. Following the same logic, we introduced one artificial root on the 50% of the SPLOT models. In order to

[1] _FeatureModelDiagnosis.pl available at: https://sites.google.com/site/raulmazo/

do that, we created a simple tool[2] that translates models from SPLOT format to facts and automate the assignation of artificial attributes, allowing repeated attributes inside each affected feature (between 1 and 5 features per affected feature), and roots. Evaluation was made in the following environment: Laptop with Windows Vista of 32 bits, processor AMD Turion 64 bits X2 Dual-Core Mobile RM-74 2,20 GHz, RAM memory of 4,00 GB and GNU Prolog 1.3.0.

A. Precision of the detection

One example of the effectiveness of our approach is the 56 conformance anomalies of the models taken from SPLOT, violating rules 2, 7 or 9. For example, in the Model transformation taxonomy feature model [35], features like `Form`, `Semantically_typed`, `Interactive`, `Source`, `Syntactically_typed`, `Target` and `Untyped` appear twice. In addition, we found 1553 conformance defects with rules 1 and 3. These came from the attributes and root features that we intentionally introduced in the SPLOT models. A manual inspection on a sample of 56 conformance defects showed that our approach identify the 100% of the anomalies with 0% false positive, as expected due to the completeness of GNU Prolog.

B. Computational Scalability

The execution times of our tool during the experiment show that our approach is able to support a smooth interaction during a conformance checking process. Indeed, each conformance rule was executed within milliseconds. Figure 3 shows the execution time of each one of the nine conformance rules in the 50 models. In Figure 3 each plot corresponds to a conformance rule: Figure 3(1) corresponds to rule 1, Figure 3(2) corresponds to rule 2 and so on. Times in the Y axis are expressed in milliseconds (ms) and X axis corresponds to the number of features.

Initial analyses showed us that 74,2% of the queries take 0 ms, which actually means that the execution time is less than 1 ms (the GNU Prolog solver does not offers times in microseconds and please note that the timer granularity of GNU Prolog under Windows is 5 ms). Give the lack of reliability of measures of very short execution times, we executed five times each of the nine rules for each of the 50 models, which means a total of 2250 (9X50X5) queries. The time measures presented in the paper are the average of the five executions of each rule on each model (450 consolidated results). In small models (9 to 100 features) the worst rule execution time was 32 ms. In large models (100 to 10000 features), execution time of each rule was less than 140 ms. The maximal time taken by the tool to execute all nine conformance rules on complete models was 265 ms (a ¼ of a second).

Table 1 shows the correlation coefficient (R^2) between the number of features in the models and the time that each rule takes to be executed. Of course, the R^2 does not prove independency between these variables. However, it gives a good indication of their dependency/independency. In the case of rules 1, 2, 3, 4, 6, and 8, the correlation coefficient is next to 0. This means that, despite the NP complexity of this kind of problems associated with CSP (Constraint Satisfaction Problems), these rules seem to be scalable to large models with the application of our approach. This is also shown in Figure 3 that indicates that every rule can be checked in a linear (seven rules) and polynomial (rules 5 and 7) time. We believe this is due to the fact that our approach does not evaluate the whole models but only the elements concerned by the rule.

Table 1. Correlation coefficients between "number of features" and "rules execution time" per each rule and over the 50 models.

Rule	1	2	3	4	5	6	7	8	9
R^2	0,01	0,15	0,01	0,04	0,74	0,02	0,87	0,01	0,35

C. Usability

Our approach was also implemented to check the conformance of DOPLER variability models. We used two industrial cases and the DOPLER variability meta-model proposed by Dhungana et al. [39]. In this experience we implemented the DOPLER meta-model, models and conformance rules using SWI-Prolog[3]. Using our approach, engineers could check conformance of product line models specified in other languages, write their own conformance rules and use another declarative language different from GNU Prolog. We believe these are important features for usability.

This paper does not address how to best visualize conformance anomalies. Much of this problem has to do with human-computer interaction and future work will study this. Another important issue that is not addressed in this paper is the downstream economic benefits. For example, one could raise the question how does fast detection of conformance anomalies really benefit software engineering at large? How much does it cost to fix an error early on as compared to later on? These complex issues have yet to be investigated.

[2] parserSPLOTmodelsToCP.rar available at: https://sites.google.com/site/raulmazo/

[3] http://www.swi-prolog.org/

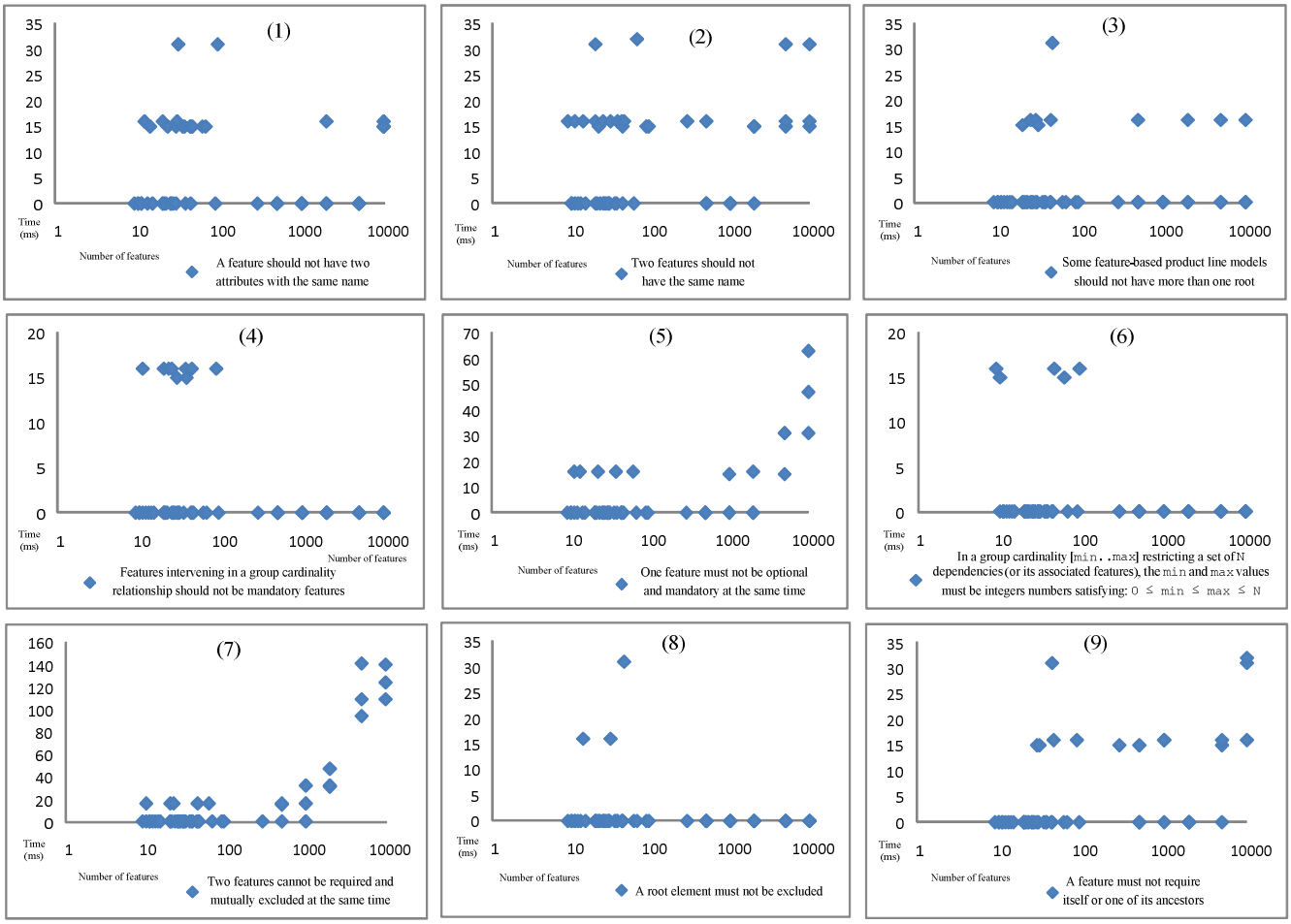


Figure 3. Execution time, of the 9 conformance rules, per number of features.

VI. Perspectives and Conclusion

In this paper we proposed a conformance checker that uses parameterizable rules to detect non-conformance in extended feature models. Our approach consists in representing FMs as CLPs, namely with sets of meta-facts. Conformance rules are parameterizable query functions expressed in a declarative manner. In addition, our experience has shown that these rules, implemented as queries, are something that a modeling tool could easily enforce. The result of the query is a collection of elements that do not conform with the meta model. As the experiment demonstrates, our approach to conformance checking is correct, useful, and our tool implementation is fast and scalable.

Future works include the following items. First, we envision to implement an incremental checker with rule scopes such as the one proposed by Egyed in [6]. We expect this will reduce the execution time of some of our conformance rules. Also, we plan to devise a classification of conformance rules according to their severity and complexity. We will explore how to fix non-conformances in an automated way. We believe that the classification can serve as a guide to define strategies to fix non-conformances and better exploit the capabilities of Egyed's incremental conformance checking technology. It is also our intention to explore the question of how to best present feedback to the engineer. The efficiency of our approach depends on how conformance rules are written, because in each rule we make explicit the elements of the model that will be evaluated. Since conformance rules are typically written manually (by engineers), it is future work to investigate how to automatically optimize conformance rules and if possible how to automatically generate conformance rules directly from the product line model's meta-model.

Our approach has been applied to feature models. However, we argue that it is also applicable to other variability formalisms (e.g. OVM, goals, UML, etc) [43]. Our experience with DOPLER already showed us that the meta-models share some common concepts such as variability and that the resulting conformance checking rules are very similar, when not identical. A significant scientific result would be to define generic rules that could be adapted to any meta-model in a fully automatic way in a similar way to Salinesi et al. [44].

ACKNOWLEDGMENTS

This research was partially funded by the Austrian FWF under agreement P21321-N15 and Marie Curie Actions - Intra-European Fellowship (IEF) project number 254965. This work was also supported by the Intra-European Fellowship "Bourse de mobilité Île de France" and the French Minister of Higher Education and Research.

REFERENCES

[1] Pohl K., Bockle G., van der Linden F.J. Software Product Line Engineering: Foundations, Principles and Techniques. *Springer* (2005)

[2] Bosch J. Design and Use of Software Architectures. Adopting and evolving a product-line approach. *Addison-Wesley* (2000)

[3] van Gurp J., Bosch J., Svahnberg M. On the Notion of Variability in Software Product Lines. Proceedings of the *Working IEEE/IFIP Conference on Software Architecture* WICSA' 01, (2001).

[4] Taylor R.N., Medvidovic N., Dashofy E. Software Architecture: Foundations, Theory, and Practice . *John Wiley & Sons* (2009)

[5] Diaz D. Codognet P. Design and Implementation of the GNU Prolog System. *Journal of Functional and Logic Programming* (JFLP), Vol. 2001, No. 6, October (2001)

[6] Egyed A. Instant consistency checking for UML. In: *International Conf. Software Engineering* (ICSE'06), pp. 381–390. ACM Press, New York (2006)

[7] Van Hentenryck P. Constraint Satisfaction in Logic Programming. *The MIT Press* (1989).

[8] Benavides D., Trujillo S., Trinidad P. On the modularization of feature models. In *First European Workshop on Model Transformation*, September (2005).

[9] Mendonca M., Branco M., Cowan D. S.P.L.O.T.: software product lines online tools. *In OOPSLA Companion*. ACM (2009), http://www.splot-research.org.

[10] Schobbens P.Y., Heymans P., Trigaux J.C., Bontemps Y. Generic semantics of feature diagrams, *Journal of Computer Networks, Vol 51, Number 2* (2007).

[11] Czarnecki K., Helsen S., Eisenecker U. W. Formalizing cardinality-based feature models and their specialization, *Software Process: Improvement and Practice,* 10 (1) pages 7–29, (2005).

[12] Salinesi C., Mazo R., Diaz D., Djebbi O. Solving Integer Constraint in Reuse Based Requirements Engineering. *18th IEEE International Conference on Requirements Engineering* (RE'10). Sydney, Australia (2010).

[13] Streitferdt D., Riebisch M., Philippow I. Details of formalized relations in feature models using OCL. In Proceedings of *10th IEEE International Conference on Engineering of Computer–Based Systems* (ECBS'03), Huntsville, USA. IEEE Computer Society, pages 45–54 (2003).

[14] Griss M., Favaro J., d'Alessandro M. Integrating feature modeling with the RSEB, in: Proceedings of the Fifth International Conference on Software Reuse, Vancouver, BC, Canada, June (1998).

[15] White J., Doughtery B., Schmidt D. Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Journal of Systems and Software*, 82(8):1268–1284 (2009).

[16] Trinidad P., Benavides D., Durán A., Ruiz-Cortés A., Toro M. Automated error analysis for the agilization of feature modeling, *Journal of Systems & Software – Elsevier* (2008).

[17] Mendonça M., Wasowski A., Czarnecki K. SAT–based analysis of feature models is easy. In Proceedings of the *Sofware Product Line Conference* (2009).

[18] Von der Massen T., Lichter H. Deficiencies in feature models. In *Tomi Mannisto and Jan Bosch, editors, Workshop on Software Variability Management for Product Derivation - Towards Tool Support* (2004).

[19] Kang K., Cohen S., Hess J., Novak W., Peterson S. Feature-Oriented Domain Analysis (FODA) Feasibility Study, *Technical Report CMU/SEI-90-TR-21, Software Engineering Institute*, Carnegie Mellon University, November (1990).

[20] Clarke E. M., Grumberg O., Long D. E. Model checking and abstraction, *ACM Transactions on Programming Languages and Systems* (TOPLAS), v.16 n.5, p.1512-1542, Sept. (1994).

[21] Thaker S., Batory D., Kitchin D., Cook W. Safe composition of product lines, Proceedings of the *6th international conference on Generative programming and component engineering*, October 01-03, Salzburg, Austria, (2007).

[22] Schulte C, Stuckey P. J. Efficient constraint propagation engines. *ACM Trans. Program. Lang. Syst*., (2008).

[23] Stahl T., Völter M., Czarnecki K. Model-Driven Software Development: Technology, Engineering, Management. San Francisco, *Wiley*, June 2006.

[24] Clements P., Northrop L. Software Product Lines : Practices and Patterns, *Addison Wesley, Reading*, MA, USA (2001).

[25] Vierhauser M., Grünbacher P., Egyed A., Rabiser R., Heider W. Flexible and Scalable Consistency Checking on Product Line Variability Models. Proceedings of the *25th IEEE/ACM International Conference on Automated Software Engineering* (ASE), Antwerp, Belgium, ACM (2010).

[26] Kim L., Klaus P. Towards automated consistency checks of product line requirements specifications. Proceedings of the *twenty-second IEEE/ACM international conference on Automated software engineering* ASE'07, USA (2007).

[27] Barachisio L., Cardoso V., de Almeida E. A Support Tool for Domain Analysis. In *4th International Workshop on Variability Modelling of Software-intensive Systems*, Linz-Austria (2010).

[28] Matthias R., Kai B., Detlef S., Ilka P. Extending feature diagrams with UML multiplicities. Proceedings of the *Sixth Conference on Integrated Design and Process Technology,* Pasadena, CA (2002).

[29] Salinesi C., Rolland C., Mazo R. VMWare: Tool support for automatic verification of structural and semantic correctness in product line models. In *Third International Workshop on Variability Modelling of Software-intensive Systems (VaMos)*, pages 173–176 (2009).

[30] Cabot J., Teniente E. Incremental evaluation of ocl constraints. In: *Dubois, E., Pohl, K. (eds.) CAiSE'06*. LNCS, vol. 4001, pp. 81–95. Springer, Heidelberg (2006).

[31] Blanc X., Mougenot A., Mounier I., Mens T. Incremental Detection of Model Inconsistencies Based on Model Operations. In *CAiSE'09*, pages 32-46, (2009).

[32] Paige R. F., Brooke P. J., Ostro J. S. Metamodel-based model conformance and multiview consistency checking. *ACM Transactions on Software Engineering and Methodology*, (2007).

[33] Olivé A. Integrity Constraints Checking In Deductive Databases, Proceedings of the *17th International Conference on Very Large Data Bases*, p.513-523, September 03-06, (1991).

[34] Kowalski R.A., Sadri F., Soper P. Integrity checking in deductive databases. In: Proc. *International Conference on Very Large Data Bases (VLDB)*, pp. 61–69. Morgan Kaufmann, San Francisco (1987).

[35] Czarnecki K., Helsen S. Classification of model transformation approaches. In Online Proceedings of the *2nd OOPSLA03 Workshop on Generative Techniques in the Context of MDA*, Anaheim, (2003).

[36] Finkelstein A.C.W., Gabbay D., Hunter A., Kramer J., Nuseibeh B. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, pages 569–578 (1994).

[37] Harel D., Rumpe B. Meaningful modeling: what's the semantics of ''semantics''? *IEEE Computer 37(10)*, pages 64–72 (2004).

[38] Salinesi, C., Mazo, R., Diaz, D. Criteria for the verification of feature models, In *28th INFORSID Conference*, Marseille, France (May 2010).

[39] Dhungana D., Heymans P., Rabiser R. A Formal Semantics for Decision-oriented Variability Modeling with DOPLER, Proc. of the *4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, Linz, Austria, ICB-Research Report No. 37, University of Duisburg Essen, 2010, pp. 29-35.

[40] J. Jaffar , J.-L. Lassez, Constraint logic programming, Proceedings of the *14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, p.111-119,, Munich, West Germany (January 1987).

[41] Lora-Michiels A., Salinesi C., Mazo R. A Method based on Association Rules to Construct Product Line Model. *4th International Workshop on Variability Modelling of Software-intensive Systems*. Linz, Austria, Janvier 2010.

[42] Djebbi O., Salinesi C., Fanmuy G. Industry Survey of Product Lines Management Tools: Requirements, Qualities and Open Issues, *International Conference on Requirement Engineering (RE), IEEE Computer Society*, New Delhi, India, October 2007.

[43] Djebbi O., Salinesi C. Criteria for Comparing Requirements Variability Modeling Notations for Product Lines. Proceedings of *4th international workshop on Comparative Evaluation in Requirements Engineering, CERE '06*, Sept. 2006.

[44] Salinesi C., Etien A., Zoukar I. A Systematic Approach to Express IS Evolution Requirements Using Gap Modelling and Similarity Modelling Techniques, *International Conference on Advanced information Systems Engineering (CAISE), Springer Verlag*, Riga, Latvia, 2004.

[45] Salinesi C., Mazo R., Djebbi O., Diaz D., Lora-Michiels A. Constraints: the Core of Product Line Engineering. Fifth IEEE International Conference on Research Challenges in Information Science (IEEE RCIS), Guadeloupe - French West Indies, France, May 19-21 2011.

[46] Budinsky F., Steinberg D., Merks E., Ellersick R., Grose T.J. Eclipse Modeling Framework. Addison-Wesley Professional, Part of the Eclipse Series series. Aug 11, 2003.

[47] Djebbi O., Salinesi C. RED-PL, a Method for Deriving Product Requirements from a Product Line Requirements Model. In: CAISE'07, Norway, 2007.