Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-Features

Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay

Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-Features Technical Report

Maxime Cordy,* Pierre-Yves Schobbens, Patrick Heymans
University of Namur, Belgium.
{mcr, pys, phe}@info.fundp.ac.be

Axel Legay
IRISA/INRIA Rennes, France.
University of Liège, Belgium.
axel.legay@inria.fr

***Abstract*—Model checking techniques for software product lines (SPL) are actively researched. A major limitation they currently have is the inability to deal efficiently with non-Boolean features and multi-features. An example of a non-Boolean feature is a numeric attribute such as *maximum number of users* which can take different numeric values across the range of SPL products. Multi-features are features that can appear several times in the same product, such as processing units which number is variable from one product to another and which can be configured independently. Both constructs are extensively used in practice but currently not supported by existing SPL model checking techniques. To overcome this limitation, we formally define a language that integrates these constructs with SPL behavioral specifications. We generalize SPL model checking algorithms correspondingly and evaluate their applicability. Our results show that the algorithms remain efficient despite the generalization.**

***Index Terms*—Software Product Lines; Numeric Features; Feature Cardinalities; Model Checking; Semantics; Tools**

I. Introduction

Software Product Line (SPL) engineering is an increasingly widespread software development paradigm in which similar software products are designed and developed as a family to make economies of scale [1]. A key challenge in SPL engineering is the management of the differences between the products, aka *variability*. Features are first-class abstractions to model and reason on variability. They specify characteristics that may be present or absent in a product. Relations between features, like parent-child, implication and exclusion, are usually captured in a feature model (FM). Since they were first introduced by Kang *et al.* in 1990 [2], FMs became more sophisticated. For detailed surveys of FM languages see, e.g., [3], [4]. In this paper we use TVL [4], one of the latest incarnations of FMs, due to some of its advantages: high expressiveness, formal semantics and tool support. A TVL excerpt is given in Figure 1. TVL and the example are properly introduced in Section II.

Many SPLs are complex critical systems and variability is known to be the source of additional complexity. Therefore, efficient quality assurance is paramount. Model checking is an established automated technique for verifying system behaviour. As for single systems, model checking techniques for SPLs are actively researched [5]–[10]. Our past work [7]–[9] was concerned with one of the major challenges in SPL model checking: as the number of features grows, the number of products increases exponentially. We thus proposed various techniques for efficient SPL model checking. We introduced Featured Transition Systems (FTS), a mathematical formalism to model the behaviour of SPL products in a concise manner [7]. We also designed efficient algorithms that identify which products exhibit undesired behaviour. However, FTS are a fundamental formalism, not meant to be used directly by engineers. We thus proposed high-level specification languages to be used on top of FTS, notably fPromela [11], an SPL-specific dialect of Promela [12]. Given the large number of different products in an SPL, it is unrealistic to describe their behaviour separately. Instead, our languages associate optional behaviour with features that must (not) be present to enable it.

A fundamental limitation of existing SPL model checking approaches, including ours, is that they do not deal with numeric features, nor features appearing several times in a product, *viz.* multi-features[1]. Numeric (as opposed to purely Boolean) features occur in FMs in the form of attributes associated with features. An example could be *Maximum number of users* which can take different numeric values across the range of products. The multi-feature construct could be used to represent, e.g., processing units which number is variable from one product to another and which can be configured independently. A recent survey, dubbed by our own experience, showed that engineers commonly need these constructs [15]. To transfer of our model checking techniques to industry, we thus have to support them. Despite evidence of their usefulness in practice, no SPL modelling tool currently supports multi-features [4], and SPL model checking tools support neither numeric nor multi-features.

In this paper, we propose a combined formalism that integrates TVL with fPromela to model the behaviour of SPLs with numeric attributes and multi-features. With the addition of attributes, optional behaviour can be made dependent not only on the presence or absence of features, but also on the satisfaction of arithmetic constraints over attributes. This implies

* FNRS research fellow

[1] Sometimes misleadingly referred to as *clones* in the literature [13], [14].

a generalization of the underlying formalism, *viz.* FTS, and its associated model checking algorithms. We implemented the complete method on top of SNIP, an FTS-based model checker [11]. The addition of arithmetic constraint solving naturally lead us to use Satisfiability Modulo Theory (SMT) solvers. Still, those are more computationally expensive than solutions for purely Boolean satisfiability. Our experiments show that while multi-feature support does not constitute a threat to performance, the use of SMT solvers to support attributes drastically increases verification time. We thus also propose an alternative solution where attributes are converted into sets of Boolean variables. This latter approach turns out to be more efficient.

Structure of the paper. Section II recalls the necessary background. In Section III we expose the challenges related to multi-features and define an extension of TVL that supports them through an array-based semantics. In Section IV, we define multi-features and attributes in fPromela. The implementation and evaluation are described in Section V. Related work is discussed in Section VI.

II. Background

The necessary concepts and background are now recalled, most of them based on our running example which is first introduced.

A. Running Example

CFDP is a highly-configurable deep-space file transfer protocol [16]. In the past, our team helped Spacebel, a Belgian company, to develop an implementation of CFDP as an SPL [17]. The original CFDP FM has 98 features. Here, we consider a small subset of the protocol, *i.e.,* the ack modes it offers. The corresponding sub-FM has 14 features and yields 1,058 different valid products. We had to limit ourselves to this subset because in addition to the variability, we had to model the behaviour of the protocol's features, which is far more complex. We did that based on the protocol specification [16] and experimented with various SPL behavioural modelling languages. This turned out to be a difficult and time-consuming activity. The resulting models describe a communication scenario where an entity (*e.g.,* a spacecraft) has to transfer a message to another one. Depending on the features of the protocol's instance in each entity, properties like successful transmission may or may not be satisfied during the transaction. The final model has 1,812,652 states.

B. TVL

Figure 1 shows an excerpt of the TVL model of CFDP. The equivalent graphical representation appears on its right. In both representations, the FM's fundamental structure remains a tree that reflects the parent-child hierarchy between the features. At the top of the tree lies the *root* feature ($CFDP$). The root is always part of a product, regardless of its other features. A feature may have *child features*; for instance, $CFDP$ has three: $Entity$, $Message$ and $Channel$. *Group cardinalities* define how many children a feature may have in any given

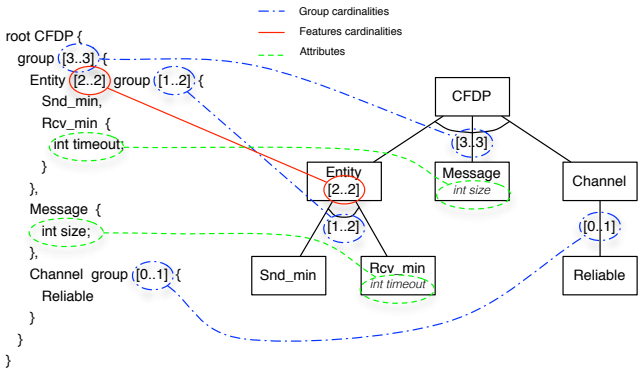


Fig. 1: A subset of the CFDP feature model, shown in TVL and in a feature diagram.

product. This construct is common in FM languages, including TVL. Here, it specifies that $CFDP$ must have exactly three children, whereas $Entity$ must have one or two.

Group cardinalities are not to be confused with *feature cardinalities* [13], [14] which specify how many instances of a feature may exist in any given product. When no feature cardinality is given, as is the case for all features in the model except $Entity$, the feature is implicitly assumed to occur at most once in each product. But whenever there is a need to allow a feature to have multiple instances, an explicit cardinality is added to the feature itself (as opposed to the group); we call such a feature a *multi-feature*. In the excerpt, the only multi-feature is $Entity$. It has a feature cardinality of exactly 2; hence two instances of this feature exist in each product. In our scenario, each instance corresponds to one of the two communicating spacecrafts. Defining $Entity$ as a multi-feature allows the two spacecrafts to bear a different configuration. If $Entity$ was a normal feature, the spacecrafts should necessarily be identical. In reality, it is more likely that they are not.

Each instance of $Entity$ must have at least one of the following child features: Snd_min (sending capabilities) and Rcv_min (receiving capabilities). Under Snd_min and Rcv_min lie additional features, which have been omitted in the figure. Features Rcv_min and $Message$ both have an integer *attribute*. The attribute $timeout$ determines the number of communication flaws that are allowed before aborting a communication while $size$ models the number of data packets that must be sent for the transmission to end. Feature $Channel$ has an optional child feature $Reliable$, which specifies whether or not the communication channel is reliable.

In addition to the specification of the tree structure, TVL allows the definition of additional constraints (*i.e.*, Boolean formulae) over both the features and their attributes (omitted in the figure). The semantics of an FM is usually defined as the set of *valid* products, *i.e.* products whose features satisfy all the constraints defined by the FM [3].

C. Featured Transition Systems (FTS)

FTS [7] is the formalism at the core of our model checking approach. Its main advantage over competing approaches is that it uses an explicit notion of feature, which brings performance improvements and allows one to relate errors and undesired behaviours to the exact set of products where they occur. FTS are directed graphs whose transitions are annotated with *feature expressions*, *i.e.* Boolean formulae defined over the set of features. For instance, the feature expression $Message \wedge Channel$ represents the set of products that have the feature $Message$ *and* the feature $Channel$; $\neg Reliable$ models the set of products that do *not* have the feature $Reliable$. A product is thus able to execute a transition iff its set of features satisfies the associated feature expression. A model-checking algorithm takes that information into account while looking for error states. It can thus keep track of which products are able to execute the currently analysed behaviour. Feature expressions constitute an intuitive and flexible way to represent variability in behavioural models. In their current form, FTS do not support expressions over multi-features and attributes. Formally, they are defined as follows.

Definition 1 *An FTS is a tuple* $(S, Act, trans, I, AP, L, d, \gamma)$*, where* S *is a set of states,* Act *is a set of actions,* $trans$ *is a set of transitions,* I *is a set of initial states,* AP *is a set of atomic propositions,* L *labels each state with the propositions it satisfies,* d *is an FM, and* γ *associates each transition to a feature expression.*

D. SPL Behavioural Specification in fPromela

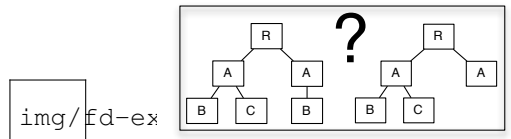
There are two kinds of approaches to implement (or model) SPLs [18]. *Compositional* methods capture the effects of features in isolated modules. A desired product is then obtained by composing the right set of modules. On the contrary, *annotative* approaches directly adorn code/models with constraints over the features, *e.g.,* feature expressions. These annotations express that parts of the code/model are exclusive to the set of products satisfying the formulae.

fPromela [11] falls into the latter category. It is an executable language based on SPIN's input syntax [12]. In an fPromela model, the behaviour of each process is described in a `proctype` structure. Within a process, executable statements are expressed using constructs inspired by imperative programming. Each of these can be annotated with feature expressions, such that only the products satisfying a formula are able to execute the associated statement. Let us consider the excerpt shown in Figure 2. Features are declared as Boolean fields of a user-defined structure called `features`. In this model, two processes of type `cfdp_entity` are specified. At some point, the specification of a `cfdp_entity` splits into two parts: one for the products satisfying the feature expression Snd_min and one for the others.

The semantics of an fPromela model is an FTS. Each process is first translated into an FTS. A state corresponds to a variable valuation and a node of execution. Transitions between states are determined according to the executable

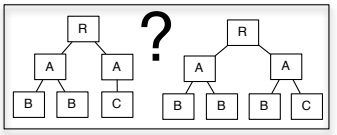
```
...
typedef features {
   ...
   bool Snd_min
}
features CFDP;
active[2] proctype cfdp_entity(...) {
   int i = 0;
   ...
   if  ::  CFDP.Snd_min -> ...
       ::  else -> ...
   fi;
   ...
}
```

Fig. 2: Partial CFDP model.



(a) FM example

(b) Group cardinalities (1)



(c) Group cardinalities (2)

(d) Feature cardinalities

Fig. 3: Ambiguities introduced by feature cardinalities.

statements. Feature expressions labelling transitions are directly derived from the fPromela model. Once all the processes have been translated, the final FTS is obtained by computing their parallel composition [11].

An fPromela model may include behaviour that is executable by no valid product. Let us suppose that during an execution, the model goes through a transition annotated by feature Snd_min then through one annotated by its negation. The resulting execution path is inconsistent because it requires both the presence and the absence of the feature. To avoid exploring this path, we use SAT solving methods which detect non-satisfiable formulae. Similarly, an execution path may require an invalid combination of features. It is consequently not a behaviour of the SPL since it cannot occur in any valid product. Such paths are avoided by checking the feature expression associated with the path of execution against the FM [11].

III. Array-Based Semantics for Multi-Features

The syntax and the semantics of attributes in TVL are already defined but the language does not support multi-features. To overcome this, we extended the syntax as follows: unless there is exactly one instance, the cardinalities are written between brackets next to the name of the feature (see Figure 1). Additional constraints over multi-features are

described further in this section. For clarity, we name TVL* our new version of TVL. Before we give it a formal semantics, we have to deal with a number of issues.

A. Challenges in the Definition of Feature Cardinalities

As explained by Michel *et al.* [14], feature cardinalities introduce semantic ambiguities. When a non-terminal feature has a maximum cardinality greater than one, two ambiguities exist regarding its group cardinalities. For instance, consider the FM in Figure 3a and the two instances shown in Figure 3b. According to the feature diagram shown in the former figure, feature A has two instances and a group cardinality of [1..2] (only one or two child features of A may exist). However, it is unclear whether the group cardinalities apply *locally* under *each* instance of A or *globally* for *all* instances of A. In other words, either each instance can have one or two child features (left diagram in Figure 3b), or there must be one or two child features of A altogether (right diagram). Since our approach is centred on multi-features, we consider the local interpretation like Michel *et al.* [14].

The second ambiguity lies in the scope of group cardinalities; see Figure 3c. In the left diagram, cardinalities restrict the number of *instances* of A's child features. On the contrary, in the right diagram, the two instances of B under the leftmost instance of A are counted once; in this case, cardinalities constrain the number of distinct child features. Like Michel *et al.* [14], we believe that the original intent of group cardinalities is to restrict the number of *features* in a product; we thus consider the second option.

Multiple interpretations are also possible for feature cardinalities. Let us consider Figure 3d. As for group cardinalities, either feature cardinalities apply globally and count the total number of instances of a feature (left diagram), or they apply locally and count the number of instances of a feature under a specific instance of its parent feature (right diagram). As before, we adopt the most local option, *i.e.* the latter.

Another issue concerns the identification of a feature by means of its name. In FMs without multi-features, the (relative) name of a feature works as a unique reference to that feature. This is not the case for multi-features. Let us consider again the TVL* model of CFDP (Figure 1). The relative name Snd_min can refer to a child feature of either the first instance of $Entity$ or the second one. We have to identify this instance using an "absolute" name (called *fully qualified*). In TVL, however, no construct exists for referring to a precise instance. Moreover, the semantics proposed by Michel *et al.* defines the children of a feature as a multiset of instances [14]. One cannot refer to a precise element of a multiset in natural language. Our SPL behaviour specification language requires this capability; the definition of Michel *et al.* is thus inappropriate for our purpose.

We propose to represent the children of a feature by a set of arrays of instances. In a given array, all the instances are from the same feature. Each of them is identified by an index, the first index being zero. For example, the Snd_min child feature of the second instance of $Entity$ is identified by the fully qualified name $CFDP[0].Entity[1].Snd\_min[0]$. When the maximum number of instances of a given feature is 1, the index can be omitted; $CFDP.Entity[1].Snd\_min$ is thus equivalent to the above name. An attribute of a given instance must be referred to using its fully qualified name as well, *e.g.*, $CFDP.Message.size$.

Since we introduce multi-features in the syntax of TVL*, we must provide means to specify constraints over them, their attributes, and their number. Fully qualified names are already suitable to refer to precise instances or attributes. However, it is currently impossible to reason over a whole array. For example, one cannot express that *the number of instances of a feature must not exceed the value of another feature's attribute*, or that *every instance of a feature must satisfy a given constraint*. To address this limitation, we define the operator `card` which, given a fully qualified multi-feature name, returns its number of instances. *E.g.*, `card(CFDP.Entity)` always returns 2. We also define two new types of constraints: `forall(m)`$\{\phi\}$ and `exists(m)`$\{\phi\}$. Intuitively, they specify that for each (resp. at least one) instance of a multi-feature `m`, the *sub-tree* of this instance satisfies the constraint ϕ. For example, the constraint `forall (`$CFDP.Entity$`)` $\{Snd\_min \vee Rcv\_min\}$ is satisfied if and only if every instance of $Entity$ has at least one child. As we will see, a notion of *context* is required for defining the semantics of such formulae.

The last challenge is related to constraints over attributes. If a feature is not part of a product, references to its attributes point to an unknown value. In this case, it is undetermined how the constraint must be evaluated. Our new types of constraints already provide a solution to that problem by specifying constraints within the context of a instance. The evaluation of a constraint is thus performed under the assumption that the instance exists within its context. Still, this issue makes such descriptions error-prone.

B. Abstract Syntax TVL*

Now that the new TVL* constructs have been informally introduced, we give them an abstract syntax and a formal semantics. Note that this abstract syntax remains valid for most feature modelling languages that support multi-features and attributes, and whose diagrams follow a tree structure.

Definition 2 *A TVL* model is a tuple* $(F, r, DE, \omega, \lambda, A, \rho, \tau, \Phi)$ *where* F *is a non-empty set of features,* $r \in F$ *is the root,* A *is the set of attributes, and:*

- $DE \subseteq F \times F$ *is the decomposition (hierarchy) relation between features. For any* $(f, f') \in DE$, f *is the parent and* f' *is the child feature. By* $children(f)$ *we denote the set of child features of* f.
- $\omega : F \to \mathbb{N} \times (\mathbb{N} \cup \{*\})$ *gives the cardinality of each feature. If* $\omega(f) = (n, m)$ *then* n *is the minimum cardinality of* f *and* m *its maximum cardinality. If* $m = *$ *then* m *can be any finite value.*
- $\lambda : F \to \mathbb{N} \times (\mathbb{N} \cup \{*\})$ *indicates the decomposition operator of a feature, i.e., its group cardinalities. It follows the same pattern as* ω.

- $\rho : A \to F$ *is a total function that returns the feature declaring a given attribute.*
- $\tau : A \to \{int, bool\}$ *assigns a type to each attribute.*
- Φ *is a Boolean-valued expression over the features* F *and the attributes* A*, expressing additional constraints.*

Furthermore, each FM must satisfy the following well-formedness rules: r *exists and is unique:* $\omega(r) = (1,1)$*; any terminal feature* f *has no child:* $\lambda(f) = (0,0)$*; DE is acyclic, that is,* $\nexists n_1, .., n_k \in N \bullet (n_k, n_1) \in DE \wedge \forall i \in \{1..k-1\} \bullet (n_i, n_{i+1}) \in DE$.

The main difference between this new syntax and that of TVL [4] is the definition of ω. In the latter, ω was meant to identify optional features. Here, we make it more general since it defines the cardinality of a feature. Although this difference might seem thin at first sight, we already showed in Section III-A that it raises a number of important issues. The definition of Φ is also different here, as we have introduced a new operator and two new types of constraints. For convenience, we suppose that attributes are either integer or Boolean. Enumeration types can be mapped to integer and are thus implicitly supported too. The introduction of real attributes in the syntax is straightforward, but requires additional type checking. The current version of our tools (presented further in Section V) does not support them.

As explained in Section III-A, only a fully qualified name can be used to identify a specific instance or an attribute. Formally, it is a tuple of the form $\big((f_0,i_0),(f_1,i_1),\ldots,(f_k,i_k)\big) \in (F \times \mathbb{N})^{k+1}$ or $\big((f_0,i_0),(f_1,i_1),\ldots,(f_k,i_k),a\big) \in (F \times \mathbb{N})^{k+1} \times A$, respectively for an instance or an attribute. A fully qualified instance name must satisfy the following. (1) The first feature is the root indexed by 0: $(f_0,i_0) = (r,0)$; (2) the decomposition hierarchy is respected: $\forall j \bullet 0 < j \leq k \bullet f_k \in children(f_{k-1})$; (3) each index satisfies the cardinalities of its associated feature: $\forall j \bullet 0 < j \leq k \bullet w(f_j) = (n,m) \Rightarrow 0 \leq i_j < m$. A fully qualified attribute name $(f_0,i_0),(f_1,i_1),\ldots,(f_k,i_k)a$ is valid if and only if $(f_0,i_0),(f_1,i_1),\ldots,(f_k,i_k)$ is valid and a is an actual attribute of f_k: $f_k \in \rho(a)$. From now on, we assume the validity of every fully qualified name.

C. Formal Semantics

The purpose of TVL is to define the set of valid products in an SPL, *i.e.* the valid combinations of features and attribute values. We mentioned before that in SPLs without attributes and multi-features, a product is uniquely identified by a set of features. For reasons explained above, this representation is not appropriate when multi-features or attributes occur in the FM. Therefore we redefine a product as a couple $(\mathcal{F},\mathcal{A})$ where $\mathcal{F} \subseteq (F \times \mathbb{N})^+$ is a set of instances and $\mathcal{A} : \big((F \times \mathbb{N})^+ \times A\big) \to \mathbb{Z} \cup \{\top, \bot\}$ is a partial function that associates attributes with values. We assume that all the attributes of any instance in $\mathcal{F}$ have a value defined by $\mathcal{A}$:

$$\forall (f_0,i_0)\ldots(f_k,i_k) \in \mathcal{F}, a \in A \bullet f_k = \rho(a) \bullet \forall\big((f_0,i_0)\ldots(f_k,i_k),a\big) \in dom(\mathcal{A}).$$

Conversely, for any attribute whose value is defined, $\mathcal{F}$ must include the corresponding feature:

$$\forall\big((f_0,i_0)\ldots(f_k,i_k),a\big) \in dom(\mathcal{A}) \Rightarrow \forall (f_0,i_0)\ldots(f_k,i_k) \in \mathcal{F}.$$

Although we could ignore these two assumptions, they simplify the definition of the semantics.

Definition 3 *Let* $M = (F, r, DE, \omega, \lambda, A, \rho, \tau, \Phi)$ *be a TVL* model and* $p = (\mathcal{F},\mathcal{A})$ *be a product. Then* p *is valid according to* M*, noted* $p \models M$*, if and only if, for* $(f_0,i_0)\ldots(f_k,i_k) \in \mathcal{F}$*:*

- $\mathcal{F}$ *contains the root:* $(r,0) \in \mathcal{F}$*;*
- $\mathcal{F}$ *respects the decomposition hierarchy:* $(f_{k-1},f_k) \in DE$*;*
- *every instance in* $\mathcal{F}$ *has its parent in* $\mathcal{F}$ *as well:* $(f_0,i_0)\ldots(f_{k-1},i_{k-1}) \in \mathcal{F}$*;*
- $\mathcal{F}$ *satisfies the group cardinalities:* $\lambda(f_k) = (n,m) \Rightarrow n \leq |\{(f_0,i_0)\ldots(f_k,i_k)(f_{k+1},0) \in \mathcal{F}\}| \leq m$*;*
- *the* i*th instance of a feature exists only if the* $i-1$*th does:* $i_k > 0 \Rightarrow (f_0,i_0)\ldots(f_k,i_k-1) \in \mathcal{F}$*;*
- $\mathcal{F}$ *satisfies the feature cardinalities:* $\omega(f_k) = (n,m) \Rightarrow \big(i_k < m \wedge n > 0 \Rightarrow (f_0,i_0)\ldots(f_k,n-1) \in \mathcal{F}\big)$*;*
- p *satisfies the additional constraints* Φ*.*

We consider that instances are contiguously placed in arrays, which simplifies the verification of multi-features and constraints over them.

Due to lack of space, we do not provide all the satisfaction rules for the additional constraints. However, we detail them for the constructs we have added. As explained above, the constraints of the form `forall(m)` $\{\phi\}$ and `exists(m)` $\{\phi\}$ introduce the notion of *context*, *i.e.*, a sub-tree of the model. For the context to be well-defined, it is sufficient to know the instance it refers to.

Definition 4 *Let* $M = (F, r, DE, \omega, \lambda, A, \rho, \tau, \Phi)$ *be a TVL* model. Then* $(f_0,i_0)\ldots(f_k,i_k) \in (F \times \mathbb{N})^+$ *is a context of* M *if it is a valid fully qualified feature name.*

The initial context is the root feature. A product $p = (\mathcal{F},\mathcal{A})$ must thus satisfy the additional constraints Φ in the context $(r,0)$: $p \models_{(r,0)} \Phi$. The context changes only when one of the above two constructs are encountered:

$$p \models_c \texttt{forall(m)}\,\{\phi\} \Leftrightarrow \forall (c,m,i) \in \mathcal{F} \bullet p \models_{(c,m,i)} \phi,$$
$$p \models_c \texttt{exists(m)}\,\{\phi\} \Leftrightarrow \exists (c,m,i) \in \mathcal{F} \bullet p \models_{(c,m,i)} \phi.$$

Here, c merely acts as a prefix for features name. References to multi-features can occur in constraints, and express that at least one instance of the feature exists. Given our convention about array contiguity, this boils down to requiring the presence of the instance at index 0: p $\models_c$ `m` $\Leftrightarrow (c,m,0) \in \mathcal{F}$. As in TVL [4], the semantics of some constraints depend on the evaluation of arithmetic expressions, variables, or operators. For our new `card` operator, the evaluation is given by: $[\![\texttt{card(}m\texttt{)}]\!]_c = |\{(c,m,i) \in \mathcal{F}\}|$. If the `card` operator is

compared to a constant value v then the constraint can be reduced to a Boolean form:

$$p \models_c \texttt{card}(m) \geq v \Leftrightarrow (c, m, v-1) \in \mathcal{F} \quad (1)$$

We will also make use of this property to reduce the cost of verifying these constraints in behavioural specifications.

IV. A Behavioural Specification Language for Multi-Features and Attributes

The semantics of TVL* allows us to determine which combinations of features constitute valid products. Still, we need a way to express the behaviour of these products in a concise (*i.e.*, not individual) manner. Based on the principles of TVL*, we generalize fPromela with feature expressions supporting multi-features and attributes. Then we can verify models of this language against properties using a model checking tool (described in Section V). We name our extension fPromela*.

fPromela* extends fPromela's syntax with more general data types for representing features. For example, the data structure defined for the *Entity* feature would be:

```
typedef _Entity {
   bool is_in;
   bool Snd_min;
   bool Rcv_min
};
```

The first field of a structure is a Boolean called `is_in`. Its truth value defines the presence or the absence of a feature. The structures can be nested. Hence if, say, *Snd_min* is not a terminal feature, we can declare it as an instance of another structure. Attributes are declared as integer fields. More generally, there exists a transformation from a TVL* model to a set of fPromela* data structures. Each data structure represents a non-terminal feature, and contains one field per attribute and child feature. The type of the latter is either Boolean or another data structure, respectively if the feature is terminal or not. When the maximum cardinality of a feature is $1 < m < \infty$, the feature must be declared as an m-sized array of its corresponding type:

```
typedef features {
   _Entity Entity[2];
   ...
};
```

Because fPromela does not allow arrays of an unknown size, the transformation does not support infinite maximum cardinality (represented as a $*$ in TVL*). However, we may assume that in real cases, an upper bound can be defined.

Unlike in fPromela, the use of nested user-defined structures is compulsory in fPromela*. Indeed, a fully qualified name is required to refer to an instance (see Section III-A). fPromela* offers a simple way to represent valid fully qualified names: references to fields of the corresponding user-defined structures, starting from that of the root feature. For example, the *Snd_min* child feature of the second instance of *Entity* is uniquely referred to by `CFDP.Entity[1].Snd_min`.

fPromela* generalizes feature expressions as well. We distinguish between *static* and *dynamic* formulae. The former specify one of the following requirements: (1) the presence or absence of a given instance (*e.g.,* `CFDP.Entity[1].Snd_min`); (2) constraints over attributes (*e.g.,* `CFDP.Message.size>0`); and (3) constraints over number of instances (*e.g.,* `card(CFDP.Entity)==2`). As in TVL*, the third type of constraints is subsumed by the first one when `card` is compared to integers (see Equation 1). We can determine the products able to execute a transition associated with a static formula without executing the fPromela* model.

On the contrary, dynamic formulae are evaluated at runtime. Typically, they define constraints over attributes and number of instances in terms of variable values. For example, one could analyse the content of a CFDP message by iterating on its size:

```
i = 0;
do  ::  CFDP.Message.size > i
          -> ... i++;
    ::  else -> break;
od;
```

Note that for a given product, the actual size of the above model (in terms of states) depends on the value of the attribute in the product. Attributes in fPromela* are thus the key for specifying variable-size models. Instead of attributes, we could also use the number of instances of a feature. In Section V, we show that it is more efficient to verify a variable-size model rather than a set of fixed-size models, *i.e.* one per possible size.

Let us illustrate the last addition of fPromela* by means of the CFDP model. As mentioned before, two processes are declared and model the behaviour of two spacecrafts. These may have different configurations. In the partial model shown in Figure 2, however, we find but an ambiguous reference to feature *Snd_min*. We have seen that the introduction of multi-features permits the distinction between the features of the two entities. Still, we need a way to associate each process with the corresponding instance of *Entity*. To that aim, we define the feature *context* of a process as an instance of a feature. We propose two constructs to associate a context to a process. The first specifies it explicitly in the `run` statement of fPromela, which is used to dynamically instantiate a new process:

```
run [CFDP.Entity[1]] cfdp_entity(...)
```

The second method consists in declaring that the number of instances of a process is equal to the number of instances of a specific feature:

```
active[card(CFDP.Entity)] cfdp_entity(...)
```

In this case, the context of each instance of `cfdp_entity` is a distinct instance of *Entity*. A process cannot exist outside its associated context. Accordingly, the first transition of a process is implicitly constrained by the formula representing its context. Once defined, contexts can act as a prefix in feature

expressions. For this purpose, one may use the keyword `this`, which points to the context of the process:

```
if  ::  this.Snd_min -> ...
    ::  else -> ...
fi;
```

References to features outside the context are still possible by using fully qualified names.

More generally, one can regard contexts as an association between an fPromela* process and a FM. Thanks to them, one can describe the behaviour of parallel processes built from *different* SPLs. Analysing the behaviour of *compositions of SPLs* rather than individual SPLs is thus made possible. However, this remains out of the scope of this paper and is left for future work.

In [11], we defined the semantics of an fPromela model as an FTS. Since apart from feature expressions, fPromela* does not differ from fPromela, its semantics can be defined in terms of FTS where feature expressions have been generalized. We call the resulting formalism FTS*.

V. Implementation and Evaluation

Since the semantics of an fPromela* is an FTS, we may reuse algorithms based on this formalism [7]–[9]. We describe how we implemented the approach on top of SNIP [11], which we further used to carry out experiments.

A. Tool Description

SNIP is a model checker based on the FTS formalism. Figure 4 gives an overview of its architecture. Given an fPromela model, the *parser* builds a program graph representing it. The *semantic engine* generates on-the-fly the FTS equivalent to the program graph. The *verifier* module checks the FTS using dedicated algorithms [7], [9]. *Feature handler* abstracts feature formulae from their representations, whereas *feature solver* checks their satisfiability. The TVL library is used to compute the feature formula representing the valid products of a TVL model.

No change to the verifier module is required since it is independent of how feature formulae are represented. On the contrary, we extended or built new variants of the other modules. We updated the parser so that feature formulae are declared and referenced as user-defined data structures. We also relaxed syntactic constraints for feature formulae and modified the construction of program graphs accordingly. Dynamic formulae are handled by the semantic engine, which builds them at runtime.

Given that more general feature formulae are allowed, the previous feature handler and checker are not appropriate anymore. Both the representation and satisfiability checking of features were based on Binary Decision Diagrams (BDDs) [19] and their implementation in the CUDD library[2]. While BDDs have proven their efficiency in SNIP [11], they are unable to represent arithmetic constraints. Multi-features alone are not a problem in this regard but feature attributes are not necessarily Boolean. Satisfiability Modulo Theory (SMT) solvers appear as natural candidates for arithmetic constraints checking. Accordingly, our new feature solver is linked to Microsoft's Z3 [20]. Z3 implements many advanced techniques and structures for SMT solving. We thus implemented a new feature handler that relies on them. Equipped with Z3, our tool is able to check the satisfiability of formulae over both the features and their attributes. However, this comes at the price of a significant reduction in efficiency, as further experiments reveal. Consequently, we developed two variants of the tool: one equipped with CUDD and the other with Z3; the former supports only multi-features whereas the latter handles attributes as well. They are included in the last version of SNIP (*http://info.fundp.ac.be/fts/snip*).

[2]vlsi.colorado.edu/~fabio/CUDD

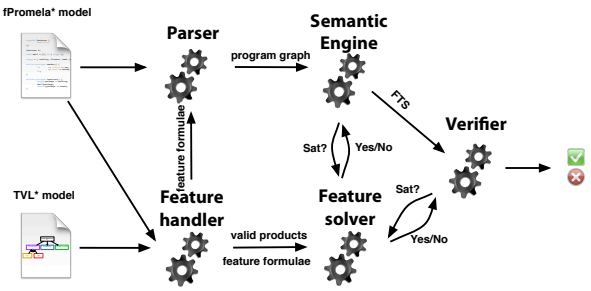


Fig. 4: Architecture of SNIP.

B. Overhead Measurement

We carried out experiments to evaluate the benefits of our approach, and to quantify the overhead that results from the additional verifications (see above) and/or the use of Z3 in place of BDD-based satisfiability solvers. The following tools are involved in the experiments: SNIP (as described in [11]), our BDD-based variant that supports multi-features (SNIP-MF), and our second variant equipped with Z3 (SNIP-Z3).

The addition of multi-features to fPromela* generates additional computations in the semantic engine of SNIP. Moreover, feature attributes require more general satisfiability-checking techniques, *i.e.,* SMT solvers. These two generalizations are likely to negatively impact the performance of our tool. To quantify the resulting loss, we compare the time needed to model check three fPromela models against five properties each. The models include neither multi-features nor attributes. They are: (1) a minepump system [7], [21] (250,561 states to explore); (2) a restricted version of CFDP where attributes have received a fixed value and the two instances of *Entity* are regarded as a single feature (1,801,581 states to explore); (3) an elevator model inspired from Plath and Ryan [22] (58,945,690 states to explore). We checked different kinds of properties including deadlock freedom, safety properties, and liveness properties. All benchmarks were run on a MacBook Pro with a 2,8 GHz Intel Core i7 processor and 8 GB of DDR3 RAM. We coded an automated script to execute them. To avoid random variations, we repeated each experiment several times

TABLE I: Verification times (in seconds).

	SNIP	SNIP-MF	SNIP-Z3
Minepump #1	2.72	3.21 (+18%)	78.22 (+2,776%)
Minepump #2	2.88	2.95 (+2%)	67.86 (+2,256%)
Minepump #3	4.90	5.47 (+12%)	88.63 (+1,709%)
Minepump #4	2.96	3.42 (+16%)	65.72 (+2,120%)
Minepump #5	9.22	9.76 (+6%)	197.43 (+2,041%)
CFDP #1	10.96	11.00 (+1%)	282.47 (+2,477%)
CFDP #2	2.23	2.41 (+2%)	63.37 (+2,741%)
CFDP #3	1.11	1.11 (+0%)	25.62 (+2,208%)
CFDP #4	3.82	4.00 (+5%)	113.36 (+2,867%)
CFDP #5	17.58	18.52 (+5%)	548.98 (+3,023%)
Elevator #1	286.50	301.10 (+5%)	TIMEOUT
Elevator #2	9.53	10.47 (+10%)	9,172.05 (+96,144%)
Elevator #3	7.07	7.90 (+12%)	3,316 (+46,799%)
Elevator #4	3.5	3.55 (+1%)	1,858.65 (+53,004%)
Elevator #5	235.67	252.94 (+7%)	TIMEOUT

and computed the average.

Results are shown in Table I. We observe that multi-feature support in our BDD-based tool generates but a small increase in verification time (between 2% and 18% for Minepump; between 0% and 5% for CFDP; between 1% and 12% for Elevator). This increment does not depend on the size of the model. In every model, the increase is less than 6% for longest-to-check properties. This indicates that the size of the property has no impact on the loss of performance either.

On the contrary, the Z3 variant suffers from a large performance drop. In Minepump and CFDP, the checking time ranges from 21 to 31 times more than for the other two tools. The tool performs even worse on Elevator: the verification time is multiplied by a factor between 468 and 962. For the first and fifth properties of that model, we could not even get accurate results; according to our estimations, verifying those properties would take 24 and 15 hours, respectively. Our SMT-based tool does not scale well with the size of the model.

We carried out further analyses to explain this phenomenon. For each tool, we computed the time required for satisfiability checking and formulae manipulation (*i.e.*, conjunction, disjunction, negation, simplification) during each experiment. In Table II we provide the time share of the two computations in percentage, for every model and property.

For SNIP and SNIP-MF, the absolute time needed for satisfiability checking (`Sat`) and formulae manipulation (`Man`) are identical. Indeed, given that all the models are free from multi-feature and attribute, the only difference between the two tools are the detection of dynamic formulae, which does not depend on `Sat` or `Man`. For these tools, it turns out that `Sat` constitutes but a small part of the total time (between 6% and 15%). Although this share tends to slightly increase with the size of the model, it is not a major threat to scalability. On the contrary, `Man` is an increasingly important part of the total time. While the time share does not grow significantly between Minepump and CFDP, it nearly doubles in the Elevator case, ranging from 38% to 53%. This indicates that *managing a large number of BDDs is the main challenge for efficient BDD-based tools*.

TABLE II: Share of verification time due to satisfiability checking and formula manipulation (in percentage).

	SNIP		SNIP-MF		SNIP-Z3	
	Sat	Man	Sat	Man	Sat	Man
M. #1	8.59	27.72	7.28	23.48	75.38	22.16
M. #2	6.07	18.26	5.93	17.83	80.37	16.82
M. #3	5.94	17.87	5.32	16.00	74.92	21.99
M. #4	6.02	18.18	5.21	15.74	80.35	17.77
M. #5	5.98	17.36	5.65	16.40	79.05	16.96
C. #1	10.14	28.79	10.10	28.69	73.07	20.86
C. #2	11.79	27.24	10.91	25.21	77.85	16.65
C. #3	8.44	20.77	8.44	20.77	77.14	15.96
C. #4	9.25	22.70	8.84	21.68	81.87	13.72
C. #5	9.38	24.32	8.91	23.09	78.57	13.49
E. #1	14.70	52.82	13.99	50.26	TIMEOUT	
E. #2	12.47	37.97	11.35	34.56	95.45	3.37
E. #3	12.09	38.23	10.82	34.21	94.68	4.87
E. #4	13.17	39.94	12.98	39.38	95.13	4.47
E. #5	11.31	37.57	10.54	35.0	TIMEOUT	

In the third tool, `Sat` is the most costly computation. Its share is already between 73% and 82% for Minepump and CFDP, and reaches astonishing proportions in the Elevator case (about 95%). This clearly shows that expensive satisfiability checking threatens scalability. We measured the average time for a `Sat` computation. It amounts to $6.59 * 10^{-4}$ seconds for SNIP-Z3 as opposed to $3.36 * 10^{-7}$ in BDD-based tools. In every model and property, the remaining computation time is due to feature manipulations for the most part. The average time of these is $1.62 * 10^{-5}$seconds – as opposed to $3.68 * 10^{-7}$ in the other tools. Together, `Sat` and `Man` always make up more than 92% of the overall verification time, and even over 95.5% in Elevator.

In our context, SMT-based solutions thus appear less efficient than BDDs. This has to be confirmed through the replacement of Z3 by other SMT solvers. Indeed, Z3 offers many facilities that we do not need. Using another solver specifically designed for our purpose might yield better results. We leave that for future work. Nevertheless, the poor performance raises the need for alternatives to SMT.

C. Explicit Attributes versus Boolean Conversion

As an alternative to Z3, we propose to transform integer attributes into a set of Boolean variables, *i.e.*, one per attribute value. This results in a model without attribute, which can thus be checked by SNIP-MF. Each time a constraint over attributes is encountered in an fPromela* model, it is first re-written in a form where only Boolean negation and greater operators occur. For instance, the constraint $CFDP.Message.size <= 3$ is converted into $\neg(CFDP.Message.size > 3)$. Then, a Boolean variable representing that constraint is created and replaces it. While this solution avoids the overhead of SMT solving, it reduces the performance of SNIP-MF because more variables have to be dealt with.

The following experiments aim at evaluating whether this is more efficient than the Z3-based solution. We consider the complete CFDP model and compare the time needed by

TABLE III: Verification times for the CFDP example (in seconds).

	SNIP-MF	SNIP-Z3	Speedup
#1	9.79	898.88	91.82
#2	2.72	345.86	127.15
#3	1.35	137.05	101.51
#4	12.24	1694.61	138.45
#5	21.44	3661.89	170.79

SNIP-Z3 and SNIP-MF to perform the verifications. Results are shown in Table III. It turns out that the transformation implemented in SNIP-MF yields far better results than SNIP-Z3. We observe a speedup ranging from 91.82 to 170.79. Interestingly, apart from Property #1 the difference in speed increases as the properties are longer to check (see in particular Property #5). This corroborates our previous results and tends to show that in our context, attributes conversion is more viable than SMT. This has to be confirmed through additional experiments with more specific SMT solvers, though.

D. Variable-Size Models

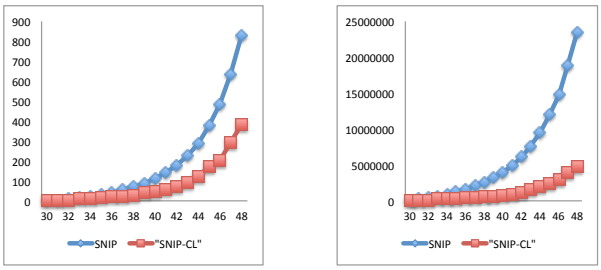
In Section IV, we mentioned that multi-features and attributes can represent variable-size models. Our new tools have thus the capability to verify in a single execution a set of models varying only by their size. They avoid redundant checking of common parts of these models, and are thus potentially more efficient. However, multi-feature support has an impact of performance. We must therefore evaluate if the avoidance of redundancy offsets the loss in efficiency.

We consider the sieve of Eratosthenes modelled in fPromela*. The size of the model is determined by the number of primes to compute. We compare the performance of (1) SNIP applied on models where the number of primes is fixed, and (2) SNIP-MF executed on a model where this number is equal to the number of instances of a feature. For a maximum number n, we compute the time needed by SNIP for successively checking models with 1 to n numbers to compute. Then, we execute SNIP-MF on the variable-size model.

Figure 5a shows the resulting verification times for n ranging from 30 to 48, whereas Figure 5b presents the number of explored states. Although the checking time always grows exponentially, it turns out that SNIP-MF performs increasingly better than SNIP. As illustrated in Figure 5b, this is because the former explores fewer and fewer states than the latter. However, we observe that for SNIP-MF, the verification time raises more rapidly than the number of explored states; the overhead due to multi-features has again a visible impact on the overall performance. Moreover, the benefit might not always exist. When the largest model has significantly more states than the smaller ones, the time needed to check these is negligible. In such cases, SNIP-MF might not be more efficient than SNIP applied on each model size.

VI. Related Work

Other SPL model checkers equipped with specification languages exist. Plath and Ryan [22] designed fSMV in the context of feature interaction detection. This is a compositional language where features are described in separate modules. We previously developed an fSMV model checker [8]. Gruler *et al.* [10] extended the CCS process algebra with a variability operator that models alternative choices. Their language is however less human-readable than fPromela. Apel *et al.* [5] developed SPLVerifier, a tool chain for product-line model checking. Features are specified in separate modules written in C or Java. The framework of Asirelli *et al.* [6] centred on modal transition systems does not include high-level specification languages. None of the above considers attributes and multi-features.



(a) Verification times (in sec.) (b) Number of explored states

Fig. 5: Benchmark results for the sieve of Eratosthenes.

A few attempts to reason on FM with attributes and multi-features exist. Czanercki *et al.* [13] define the semantics of an FM with feature cardinalities as the semantics of a context-free grammar derived from the FM. Michel *et al.* [14] defined another semantics based on multisets. Czarnecki and Kim [23] argue that OCL is suitable for expressing additional constraints over multi-features. Mazo *et al.* [24] use constraint logic programming to reason on FMs with multi-features and attributes. However, their procedure relies on a heavyweight transformation from visual FMs to XML, then to Gnu-Prolog's input syntax. Zhang *et al.* [25] propose a BDD-based approach for reasoning over multi-features. However, it does not support attributes and does not rely on an array-based semantics. A broader survey of feature cardinality analyses is found in [14].

SMT solvers are increasingly used in model checking. They have proven their usefulness in bounded model checking and verification of infinite-state and array-based systems [26]. We showed that SPL model checking is another interesting application domain for SMT solvers. However, they have to be combined with heuristics in order to provide an acceptable level of performance in our context.

VII. Conclusion

We dealt with feature cardinalities and numeric features in SPL model checking. Multi-features and feature attributes have received little support, be it in FMs or behavioural specification languages. These constructs pose many problems of both theoretical and technical nature. Nonetheless, the definition of languages supporting them is essential. Both constructs are useful in terms of expressiveness, conciseness, and readability. In a behaviour specification, they allow the

definition of variable-size models. Our experiments showed that it is easier to verify such models than successively checking the same model with different sizes. Multi-features are also useful for specifying parallel processes that differ by their configuration. To deal with them, we introduced the notion of feature context for parallel processes. More generally, contexts can relate parallel process with *different* FMs. Combined with feature-model composition [27], this result opens the way for behavioural verification of SPL compositions. This is an interesting direction for our future work.

Handling attributes in SPL model checking is difficult. SMT solvers appeared as natural candidates to represent and reason over numeric feature formulae. However, our experiments with Microsoft's Z3 revealed that the overhead of SMT satisfiability checking is too important, which forced us to consider alternatives like converting attributes into Boolean variables. Yet, we need but a small subset of Z3's capabilities. Another solver optimised for our purpose could perform better. A complementary solution is not to check feature formulae each time it is needed. This leads to exploring more states than necessary but reduces the number of calls to the solver.

Moreover, Boolean conversion is not always practical. The behavioural models we considered do not include quantitative aspects like real-time or cost/rewards. Feature attributes in these contexts likely represent more complex forms of variability, *e.g.*, data throughput, processor speed, or energy consumption. Model checking such SPLs requires the combination of this work with quantitative formalisms like Featured Timed Automata [28]. This is non-trivial; several theoretical issues like decidability are expected. On the technical side, the use of SMT solvers might be a mandatory step. Still, it constitutes an interesting problem and we will most likely pursue our work in that direction.

Acknowledgements

We thank Philippe Warnon and Raphael Michel for their suggestions regarding TVL*'s syntax, as well as Andreas Classen for proofreading. This work was funded by the Fund for Scientific Research – FNRS in Belgium (project FC 91490).

References

[1] K. Pohl, G. Böckle, and F. van der Linden, *Software product line engineering - foundations, principles, and techniques*. Springer, 2005.

[2] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Tech. Rep. CMU/SEI-90-TR-21, 1990.

[3] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps, "Feature Diagrams: A Survey and A Formal Semantics," in *RE'06*, 2006, pp. 139–148.

[4] A. Classen, Q. Boucher, and P. Heymans, "A text-based approach to feature modelling: Syntax and semantics of TVL," *SCP*, vol. 76, pp. 1130–1143, December 2011.

[5] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer, "Feature-interaction detection using feature-aware verification," in *ASE'11*. IEEE, 2011, pp. 372–375.

[6] P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi, "Formal description of variability in product families," in *SPLC'11*. Springer-Verlag, 2011, pp. 130–139.

[7] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," in *ICSE'10*. ACM, 2010, pp. 335–344.

[8] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic model checking of software product lines," in *ICSE'11*. ACM, 2011, pp. 321–330.

[9] M. Cordy, A. Classen, G. Perrouin, P. Heymans, P.-Y. Schobbens, and A. Legay, "Simulation-based abstractions for software product-line model checking," in *ICSE'12*. IEEE, 2012, pp. 672–682.

[10] A. Gruler, M. Leucker, and K. Scheidemann, "Modeling and model checking software product lines," in *FMOODS'08*. Springer, 2008, pp. 113–131.

[11] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens, "Model checking software product lines with SNIP," *STTT*, vol. 14, no. 5, pp. 589–612, 2012.

[12] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[13] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Formalizing cardinality-based feature models and their specialization." *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.

[14] R. Michel, A. Classen, A. Hubaux, and Q. Boucher, "A formal semantics for feature cardinalities in feature diagrams," in *VaMoS'11*. New York, NY, USA: ACM, 2011, pp. 82–89.

[15] A. Hubaux, Q. Boucher, H. Hartmann, R. Michel, and P. Heymans, "Evaluating a textual feature modelling language: four industrial case studies," in *SLE'10*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 337–356. [Online]. Available: http://dl.acm.org/citation.cfm?id=1964571.1964603

[16] Consultative Committee for Space Data Systems (CCSDS), *CCSDS File Delivery Protocol (CFDP): Blue Book, Issue 4*. NASA, 2007.

[17] Q. Boucher, A. Classen, P. Heymans, A. Bourdoux, and L. Demonceau, "Tag and prune: A pragmatic approach to software product line implementation," in *ASE'10*. ACM, 2010, pp. 333–336.

[18] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *ICSE'08*. New York, NY, USA: ACM, 2008, pp. 311–320.

[19] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys*, vol. 24, no. 3, pp. 293–318, Sep. 1992.

[20] L. De Moura and N. Bjørner, "Z3: an efficient smt solver," in *TACAS'08*. Springer-Verlag, 2008, pp. 337–340.

[21] J. Kramer, J. Magee, M. Sloman, and A. Lister, "Conic: an integrated approach to distributed computer control systems," *Computers and Digital Techniques, IEE Proceedings E*, vol. 130, no. 1, pp. 1–10, 1983.

[22] M. Plath and M. Ryan, "Feature integration using a feature construct," *SCP*, vol. 41, no. 1, pp. 53–84, 2001.

[23] K. Czarnecki and C. H. P. Kim, "Cardinality-based feature modeling and constraints: a progress report," in *International Workshop on Software Factories at OOPSLA'05*, ACM. San Diego, California, USA: ACM, 2005.

[24] R. Mazo, C. Salinesi, D. Diaz, and A. Lora-Michiels, "Transforming attribute and clone-enabled feature models into constraint programs over finite domains," in *ENASE'11*, 2011, pp. 188–199.

[25] W. Zhang, H. Yan, H. Zhao, and Z. Jin, "A bdd-based approach to verifying clone-enabled feature models' constraints and customization," in *ICSR'08*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 186–199.

[26] S. Ghilardi and S. Ranise, "Mcmt: A model checker modulo theories," in *IJCAR'10*, 2010, pp. 22–29.

[27] M. Acher, P. Collet, P. Lahire, and R. B. France, "Composing feature models," in *SLE'09*, 2009, pp. 62–81.

[28] M. Cordy, P. Heymans, P.-Y. Schobbens, and A. Legay, "Behavioural modelling and verification of real-time software product lines," in *SPLC'12*. ACM, 2012.