**Fixture**

*A Tool for Automatic Inconsistencies Detection in Context-aware SPL*

Paulo Alexandre da Silva Costa, Fabiana Gomes Marinho, Rossana Maria de Castro Andrade* and Thalisson Alves Oliveira

*Group of Computer Networks, Software Engineering and Systems (GREat), Federal University of Ceará, Fortaleza, Ceará, Brazil*

Keywords: Software Product Line, Context-awareness, Feature Model, Automatic Verifying Tool, Ubiquitous Software, Software Adaptation.

Abstract: Software Product Lines (SPLs) have been used to provide support to the development of context-aware applications, which use context information to perform customized services aiming to satisfy users needs or environment restrictions. In this scenario, feature models have been also used to guide product adaptation process and to enable systematic reuse. However, a side effect of using those models is the accidental inclusion of inconsistencies that may imply in several errors in the adapted products. Moreover, context-aware applications are exposed to a contextual change flow, which increases the occurrence and effects of such errors. Therefore, mechanisms to check the inconsistencies are necessary before they become errors in the adapted product. Nevertheless, a manual checking is highly error prone. In particular, there are inconsistencies that can be detected only when they arise due to a specific adaptation. For those reasons, it is essential to identify errors in the context-aware feature model before they yield incorrect adapted products. In this work, we present an Eclipse-based tool that supports the software engineer in the design of context-aware feature models and provides a simulation process to allow anticipating inconsistencies related to the adaptations.

1 INTRODUCTION

A Software Product Line (SPL) is a set of software intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment and that are developed from a common set of core assets in a prescribed way (Clements and Northrop, 2001). In a SPL, a software is originated from a process called product derivation. To be cost effective, a high amount of possible products is expected.

A feature model (Kang et al., 1998) is often used to represent the similarities and variabilities of an SPL in a tree-like data model (Czarnecki and Eise- necker, 2000). Moreover, well-formedness and composition properties can be defined over these feature diagrams. When they are specific to certain feature diagrams, they are called composition properties and when they are general purpose, they are just called well-formedness properties. In this work, the combination of a feature diagram with composition properties is called system model.

In that sense, a recurrent problem in SPL is to ensure that a feature model and their products obey composition and well-formedness properties (Mar-inho et al., 2012). On the other hand, the growing popularity of handheld devices created a new kind of applications that quickly react to monitored contextual information.

We call those applications context-aware, since they are aware of their situation in physical, virtual and user environment, and can self-adapt, benefiting from knowledge of the situation (Poslad, 2009). Those self-adaptations are called automatic contextual reconfiguration since they are solely based on observation of the environment context and do not require user attention (Schilit et al., 1994).

An SPL provides a reference architecture for the application and can support the definition of constraints that outline the application’s structure, therefore, it can be used to support adaptation of context-aware applications. The development of context-aware applications using SPL has been corroborated by the literature (Hallsteinsen et al., 2008; Lee and...
Kang, 2006; Marinho et al., 2013; Acher et al., 2009).

In this case, SPL are called CASPL (Fernandes and Werner, 2008). A CASPL defines a data model for the context environment, called context diagram, and context properties to establish when and which adaptations will be performed in the context-aware application. Similarly, well-formedness properties are also defined for context models. When manually configured, a product of an SPL can contain errors that go unnoticed by the software engineer. In context-aware applications an augmentation of product adaptations occurs. Therefore, in the domain of CASPL, checking if every possible product is in agreement with the product line feature model must be ensured.

In order to achieve efficiencies and to prevent errors, the accordance between feature model and product must be performed following a process. In that sense, a process is proposed in (Marinho et al., 2011) that comprises verification and validation techniques in five levels: feature diagram verification, composition properties verification, context diagram verification and, finally, product verification. As said before, from a single set of feature and context diagrams and composition and context properties, it is possible to configure several products. Since, this process performs validation at product level, an initial idea could be checking every possible derive product using the process as a guide. However, executing this process manually is tiresome. Furthermore, due to the number of possible derive products \(O(2^n)\), this individual checking is impracticable.

Considering this scenario, the major contribution of this paper is to check context-aware feature models and respective products correctness automatically. We call this feature model CAFM. CASPL are enhanced with a context model that comprises a context diagram, well-formedness properties for this diagram and context properties.

In this sense, we propose Fixture, a tool which corresponds to the implementation of a process previously conceived in (Marinho et al., 2011). This five-level process checks models, properties and reconfigurations of a CAFM. Additionally, the original mechanism proposes a simulation process to detect potential inconsistencies that may go unnoticed even after the three initial stages. In order to detect these errors at development time, Fixture implements this simulation process based on product states.

This paper is organized as follows. Section 2 examines the meta-models created to represent the system and context-aware parts of a context-aware application. Section 3 and Section 4 deals with the lifecycle of CASPL and how to use an automaton to perform it. Section 5 shows how we performed a case study in order to test the feasibility of the approach. Section 6 brings some related work and compares it with our approach. At last, Section 7 summarizes the paper and suggests probable future work.

## 2 META-MODELS FOR CASPL

Context-aware system models define how to represent contexts and how to support an operational life-cycle using context-aware systems (Poslad, 2009). The way the models are designed enables software engineers to conceive specific-domain properties.

In our work, context-aware applications are composed by a context model and a system model. Each of them comprises a diagram and properties (without loss of sense, properties can be seen as rules). Furthermore, diagrams and properties have a graphical representation in a tree-like structure (see Fig. 1 - Fig. 7). Associated to each of diagram and properties, there is a meta-model that represents system and context parts.

In Fig. 1, some elements from the system meta-model are depicted. This model is sufficiently expansible for defining a feature diagram as defined in (Czarnecki and Eisenecker, 2000). It's worth noting that our meta-model supports feature attributes that enables the software engineer to write specific properties involving the attributes.

The Feature meta-class is used to represent features in the system model design. Mandatory Feature, Root Feature, Optional Feature and Variation Feature meta-classes inherit from Feature meta-class. Feature meta-class has an self-relationship, representing that a feature can be associated with another feature. Feature also has a relationship with attribute that indicates that a feature can have zero or more attributes.

By analyzing the relationships, it becomes clear that it is possible to create several relationships between instances of these meta-classes. For example,
it is possible to create a relationship where an instance of Optional Feature meta-class is child of an instance of a Mandatory Feature meta-class.

In other hand, it is possible to create some undesired relationships too. For instance, a relationship between instances of Mandatory Feature and Optional Feature meta-classes where Mandatory Feature instance is child of Optional Feature instance is not appropriate. Since an instance of Optional Feature meta-class can not be present in a derivate product, its Mandatory Feature instance child would not be present either. An instance of a Mandatory Feature meta-class being not present in a derivate product is a clear violation of the Mandatory feature definition (which claims mandatory features are present in every derivate product).

In this scenario, we define well-formedness properties as conditions that must be satisfied by instances of meta-models presented in this work. In the previous paragraph, it is possible to enounce a well-formedness property: "A mandatory feature can not be child of an feature that is optional or a feature that is child of an optional feature". This well-formedness property example is checked against system model, but other well-formedness properties are checked against the context model, composition properties and context properties. Regardless the relationships the meta-models define, the well-formedness must be obeyed. It is possible to claim these well-formedness properties are domain-free. This work uses (Marinho et al., 2011) as a guide to define well-formedness properties.

The tree structure representing context diagram has three levels: context root, context entity, and context information, regardless the context domain to be modeled (see Fig. 2).

![Figure 2: Meta-model: context diagram.](image)

System models have composition properties (or composition rules) and context models have context properties. In the former, composition properties define constraints about the system model and the product reconfiguration that must be respected. In the latter, they are responsible for guiding the application adaptation when context changes take place.

An important meta-model requirement is the representation of composition properties and context properties. Both type of properties are modeled as logical implications (if...then). Their representations can be seen in Fig. 3 and in Fig. 4. In Fig. 3, the Context Property meta-class has a relationship to the Event meta-class, which represents the event that may cause a product adaptation, and a relationship to the Action meta-class, which represents the action that must be executed when the event is evaluated to true. In context properties, the antecedent is comprised of conjunctions or disjunctions over the context information values. The consequent is comprised of conjunctions dealing with the presence or absence of optional features and the features attributes. In this way, the antecedent works as an event condition and the consequent as the action to be executed if the event is triggered. In the literature, this kind of property is called ECA (event-condition-action) rule.

In Fig. 4, the Composition Property meta-class has two relationships to the Antecedent meta-class, which represent the antecedent and the consequent expressions of a composition property. Antecedent meta-class also is related to the Logical Expression meta-class, representing the left and right arguments of a composition property.

The meta-models in Fig. 1, Fig. 2, Fig. 3 and Fig. 4 are improvements of the meta-model presented in (Marinho et al., 2011) since they offer more flexibility in feature diagrams and properties creation.

Fig. 5 presents an example of a system model and Fig. 6 presents an example of context model. In Fig. 5, the system model is composed by a root feature, which comprises four features: o2, Variation v1 (OR), m1 (mandatory), and o1. Variation v1 has four variant features (o3, o4, o5, and o6). Feature m1 is mandatory and is implemented by the Attribute attr2 and feature m2 has the attribute feature Attribute attr1. Finally, feature o1 is composed by feature o7 and feature o8 and feature o7 is implemented by feature o9. In Fig. 6, the context model is composed by a Context root, which has four context entities: Context Entity ent1, Context Entity ent2, Context Entity ent3, and Context Entity ent4, and each context entity has one context information. Additionally, Fig. 7 and Fig. 8 shows examples of composition and context properties, respectively.

These four meta-models are connected by the following way: composition properties are logical implications defined over elements from the system model. Fig. 7 says if optional feature o4 is absent and optional feature o6 is present, then attr2 attribute’s value...
Figure 3: Meta-model: context properties.

Figure 4: Meta-model: composition properties.

must be greater than 25 and less than 50. Moreover, context properties connect system and context model. Context properties specify that if a specific combination of values of the context elements (defined in context model) are achieved, then the derivate product (the instance of system model) should be changed in accordance to what the context properties triggered define. This interconnection between the models are taken into account specially in the simulation phase as it will be depicted in Section 4.5.

3 LIFE-CYCLE OF CONTEXT-AWARE SYSTEMS

The life-cycle of context-aware systems shows a constant change in their structure. These changes are caused by certain changes in context information. While in real scenarios, these changes are created measuring sensors, in our work, in order to simulate these changes, we use a simulation process to create scenarios. A scenario consists of the actions that are executed. Since more than one context property can be triggered at any time, we need to know all possible combinations of triggering.

In a first thought, an idea would be taking randomly a range of values as large as possible and check them against the context properties in order to observe if they are triggered. However, this kind of approach can be time consuming and inefficient. Then, our approach focuses on choosing the values that triggers each context properties. For this, we consider the atomic formulas displayed in Table 1. We have six atomic formulas and six values (3, 15, 20, 30, 40, 60). Each value creates a range where each atomic formula can be evaluated as true or false. So, besides these original values, new values concerning the gaps between the original values should be considered. According to Algorithm 1, these new values are: 1, 7, 17, 25, 35, 45, 120.

Therefore, after checking the atomic formulas against those values (both original and new), we have the combination of atomic formulas that can be seen in Table 2. In this same table, it is worthy noting that there are repeated combinations (highlighted tuples) such that they can be removed in order to avoid unnecessary tests. After that, each combination of atomic formulas is checked against each context property event and then the relative actions that are triggered are reported. For instance, for three context properties events in Fig. 9, it will be known that they were triggered in the following way: (ARule1, ARule2), (ARule2, ARule3), (ARule1, ARule2, ARule3) and (ARule2).

The identified combinations represent the contexts and they will be used in the life-cycle of context-aware systems simulation.

The next phase is comprised of starting the simulation process.
4 AUTOMATIC INCONSISTENCIES DETECTION

4.1 Computable Form

In this work, we use a graphical notation to represent context and system models. This graphical notation is a representation of a tree-like data model. This data model allows the use of OCL-like validation. OCL-like validation is suitable to check validation conditions on a product at time. But, as aforementioned, a SPL may generate an amount of products that makes individual OCL-like validation not doable.

Against this limitation, a more powerful representation is needed. In this work, this is achieved by model transformation. We choose to translate the meta-models in propositional formulas. Another options are practicable like translating the meta-models in a constraint satisfaction problem (CSP) and then solving it aided by a CSP solver. Given the fact that our meta-model deals with composition and context properties, which are logical implications, choosing translation to propositional formulas appeared to be more straightforward. Therefore, we can use a BDD solver to perform computations over the meta-models once they are translated into propositional formulas.

Firstly, we are able to know if a system model (diagram and specific properties) makes it possible to configure at least one product. Secondly, we are able to determine if the system model allows the derivation of unsafe products. Unsafe products are diagram instances that obey composition properties but do not obey the well-formedness constraints. At last, we can identify anomalies as false optional and dead features described in (Benavides et al., 2010).

The first step is achieved by checking the propositional formula against a BDD solver. If it provides any solution to the formula, we can affirm that at least one product is configurable. The second step is achieved trying to solve the propositional formula negation. If the formula negation is satisfiable, we can affirm the CASPL can generate unsafe products. Consequently, as the negation is a logical contradiction, the CASPL does not contain unsafe products.

Additionally, the first-order based model allows the simplification of the amount of possible products. This is achieved by using a special boolean value *don’t care bit*. This little addition fits the meaning of partial configuration present in (Benavides et al., 2010). With this bit, some parts of system diagram can be momentarily ignored and resolved in a latter moment. This also allows to virtually reduce the amount of possible products what it is a great advantage of the tool.

Before using propositional formulas, it is needed to create boolean variables to use in propositional formulas. After creating these variables, we will be able to perform computations.
4.2 Variables

For each optional, variation point and mandatory feature in the system diagram, we create a boolean variable. This boolean variable represents whether the feature is present or not (the boolean variable for mandatory feature has always the value true). Moreover, the same boolean variables are used in well-formedness constraint.

The creation of boolean variables to represent attributes is complex. Instead of transforming every attribute, we will transform only the attributes that are used in composition properties. Since composition properties must be checked (and obeyed by all derivate products), the attributes used by them must be computable. Then, for each atomic formula that references an attribute, we create a boolean variable.

It is worthy noting that variables creation for feature nodes ensures that a feature node creates one and only one boolean variable (thinking in terms of math functions, it is a injective function). However, the variables creation for attributes allows the creation of more than one boolean variable. When multiple variables are created for the same attribute node, a interdependent constraint problem arise.

To exemplify the boolean variables creation process, consider the instances system model and composition properties in Fig. 5 and in Fig. 7 respectively. In Fig. 5, for each feature node (m1, m2, v1, o1,o2, ..., os) we create a boolean variable (we adopt the following notation N:node where node is the name of feature node). For instance, the o1 feature node creates the boolean variable N:o1 and so on. In Fig. 7, we create two boolean variables: af1;m1,attrv2 > 25 and af2;m1,attrv < 50. Now, we are ready to show how to compute over these variables.

4.3 Computation and Model Abstraction

(Benavides et al., 2010) has provided a compilation of the computations to ensure the well-formedness properties. However, none of them addresses computations over attributes of a feature. Since our meta-model provides features attributes, additional treatment must be addressed in order to deal with it. This additional treatment deals with the range constraint between the boolean variables created to feature nodes.

In Fig. 7, we see two atomic formulas that deals with attribute attr: which we call af1 and af2 to avoid repetitions. af1 says attr must be less than fifty and af2, greater than twenty-five. It is easy to see that it is not possible af1 and af2 be false simultaneously. While it is easy to identify violated restrictions involving only two atomic formulas referring to the same attribute feature node, the same could not be said when several atomic formulas are taken into account. For that reason, we show an algorithm to solve this problem.

For this, let’s consider a more complex example in Table 1. The algorithm consists of creating new values and then check each atomic formula against each value. Table 2 depicts this process (highlighted tuples are repeated tuples). The Table 1 contains the intuitive idea that it is not possible both af1 and af3 be false and it also calculates the other restrictions that are hard to see.

The process of creating these new values is shown in Algorithm 1, as follows.

Algorithm 1: Creates new values to be checked.

<table>
<thead>
<tr>
<th>Inputs: ordered values V</th>
</tr>
</thead>
<tbody>
<tr>
<td>newValues ← first value from V divided by 2</td>
</tr>
<tr>
<td>for all value f in V do</td>
</tr>
<tr>
<td>newValues ← value</td>
</tr>
<tr>
<td>newValues ← (value + next value)/2</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>newValues ← (last value) × 2</td>
</tr>
<tr>
<td>return newValues</td>
</tr>
</tbody>
</table>

With the values from Table 2, we create the propositional formulas displayed in Table 3. They are put...
together using the XOR operator indicating only one propositional formula must be true at the same time.

Finally, the propositional formula representing the feature diagram and composition properties is checked against a BDD. It is worthy to mention that a BDD solver uses a “do not care” bit when it is not needed to take into account a variable value to solve a propositional formula. This representation is responsible to enable less bits to represent a bigger set of possible products.

### 4.4 Automaton for Simulation

To perform the simulation, a NFA is depicted. As can be seen in (Sipser, 2006), an automaton is a 5-tuple $M$:

$$M = (Q, \Sigma, \delta, q_0, F)$$  

(1)

where $Q$ represents the finite set of states, $\Sigma$ represents the finite set of input symbols plus the empty string $\emptyset$, the transition function $\delta$, the initial state $q_0$, and the finite set of final states $F$.

To use an NFA in the simulation process, we must adapt it. The finite set of states is represented by the products of a CASPL. The initial state is a given product meanwhile the finite set of final states is comprised by unsafe products. A product said to be inconsistent when it does not obey any well-formedness property.

In a SPL with $n$ optional nodes, the total of derivate products to be examined is $O(2^n)$ which, clearly, impedes the checking of each possible product. For this same reason, in this adapted automaton the whole finite set of states is not known in advance. Although the exact finite set of states is not known, we can say the finite set of final states is comprised of inconsistent derivate products. As a matter of fact, once a product is recognised as unsafe, no more adaptations will be performed on this product. As a consequence, any inconsistent product is regarded as a final state.

The transition function $\delta$ has as input a subset $r$ where $r \in \mathcal{P}_{\geq 1}(R)$. Differently from classical automaton, the transition function in our automaton does not consider the current state. Actually, the transition can occur in any product.

The output of the transition function $\delta$ is composed by actions of each rule inputed in the transition.

The whole function $\delta$ is depicted as follows:

$$\delta : ss \in \mathcal{P}_{\geq 1}(R) \leftrightarrow ss \cdot \text{actions}$$  

(2)

When a transition takes place, the actions must be performed on the current product adapting it.

### 4.5 Simulation Algorithm

The simulation objective is to identify the bigger finite set of final states (as well as, the respective transitions) as possible. The bigger is this set, more errors and more about the life-cycle of the derivate product can be known.

As said before, the whole set of states is $O(2^n)$. To examine each derivate product is an expensive and time consuming task. In this sense, an hybrid approach heuristic could be more suitable and achieve good results.

This heuristic verifies some conditions that will break some well-formedness and composition properties regardless the current product configuration. It also verifies specific product configurations in pursuit of specific invalid product configurations.

This heuristic comprises three steps.

1. Determine which context properties can be activated simultaneously and group them in sets.
2. For each subset detected in step one, verify if any of them violates any composition property regardless the current product configurations.
3. Execute simulation in each subset detected in step one.

Step one will analyze all the context properties the user had defined and will group context property’ actions that are triggered simultaneously. There is a reason to know that: once it is known which actions can be simultaneously triggered we will use these groups to stimulate the initial configuration product.

The Algorithm 2 uses a Binary Diagram Decision (BDD) to achieve the desired output. It creates a propositional comprised by disjunction of each context property event. Once this great disjunction is created, the BDD solves this propositional formula. Next, every solution to this disjunction is checked against every event. If a solution satisfies two or more context properties it is safe to say these two or more context properties are triggered simultaneously. The output of step one is a dictionary key-value where the key is a solution found by the BDD solver and the
value is the set of context properties triggered simultaneously. Step two uses the values of this dictionary.

Algorithm 2: Calculate which context properties are activated simultaneously.

Require: solver BDD, props Property(event, action)
Ensure: \( y = x^n \)

\[
d = \text{false}
\]

\[
\text{for all } p \text{ in props do}
\]

\[
d' = d \lor p\text{.event}
\]

\[
\text{end for}
\]

\[
m: \text{Map} = \text{BDD}\text{.Solve}(d)
\]

\[
\text{for all } \text{Solution sol in } m \text{ do}
\]

\[
\text{if sol activates p.event then}
\]

\[
m\text{.get(sol)} \text{ adds prop}
\]

\[
\text{end if}
\]

\[
\text{end for}
\]

Algorithm 3: Calculate which actions are executed simultaneously.

Require: Algorithm one output

Ensure: unsafeSets

\[
\text{sets = Algorithm One Output}
\]

\[
\text{M:Map}
\]

\[
\text{for all set s in } \text{sets do}
\]

\[
\text{M\text{.get(set)} \text{ adds cp.action}}
\]

\[
\text{end for}
\]

\[
\text{return M.values}
\]

Step two verifies if a set of actions breaks any composition property regardless the current product configuration. Consider the following composition property CR: If (Storage is present \lor Wifi Connectivity is absent) then Record Movies is enabled. If a subset ss contains the following actions: Include Storage, Remove Record Movies, then it is known in advance that this subset ss will generates an unsafe configuration regardless the product.

Additionally, consider another subset of actions, \( ss' = \{ \text{Include Storage} \} \). Concerning the composition property just mentioned, to determine if \( ss' \) breaks the rule it is necessary to check it against the current product configuration. For instance, if the current product configuration \( cur \) does not contain the feature Record Movies then \( ss' \) breaks the composition property, otherwise, the actions in \( ss' \) when applied over the configuration \( cur \) do not generate an unsafe transition.

While the first type of actions is known to break a specific composition property, the second type makes the application break a composition property just in specific cases. So, the simulation process is needed to detect this kind of situations since they can not be predicted.

Algorithms 2, 3, 4 are used in Step two.

Algorithm 4: Calculate which context properties break well-formedness rules regardless product configuration.

\[
\text{sets = Algorithm One Output}
\]

\[
\text{unsafeSets}
\]

\[
\text{for all set s in } \text{sets do}
\]

\[
\text{Certificate C = actions2Certificate(set)}
\]

\[
\text{for all Composition Property cp in CR do}
\]

\[
r = \text{BDD}\text{.Solve(cp, C)}
\]

\[
\text{if } r = \text{false then}
\]

\[
\text{unsafeSets adds set}
\]

\[
\text{end if}
\]

\[
\text{end for}
\]

\[
\text{return unsafeSets}
\]

Finally, step three starts the simulation. As aforementioned, some scenarios that break any composition properties or well-formedness rules can be predicted. However, another scenarios can not be predicted since they depend on the current product configuration. The existence of these another scenarios demand an approach that can detect them. In this work, we propose the use of a simulation process aided by an enhanced automaton.

In this work, we propose two behaviors to the automaton. The first behavior is depicted in Algorithm 5 and the second one in 6.

Algorithm 5: Simulates contextual adaptations in a SPL.

Require: transition function \( \delta \), initial State \( p_0 \), maximum amount of transitions \( MAX \)

\[
q \leftarrow 0
\]

\[
\text{lastSafeState} \leftarrow p_0
\]

\[
\text{while } q \leq MAX \text{ do}
\]

\[
\text{transition} \leftarrow \text{randomly choose an transition from } \delta
\]

\[
\text{newState} \leftarrow \text{apply the actions over } \text{lastSafeState}
\]

\[
\text{if transition break any rule then}
\]

\[
\text{report transition, lastSafeState, newState as unsafe}
\]

\[
\text{else}
\]

\[
\text{report transition, lastSafeState, newState as safe}
\]

\[
\text{lastSafeState} \leftarrow \text{newState}
\]

\[
\text{end if}
\]

\[
q \leftarrow q + 1
\]

\[
\text{end while}
\]

\[
\text{return unsafeSets}
\]

The Algorithms 5 and 6 outputs are respectively depicted in Figs. 10 and 11. In both Figures, inconsistent derivate products are red marked.
Algorithm 6: Algorithm for the second behavior.

Require: automaton A, initial product \( P_0 \), maximum amount of transition \( qtd \), maximum amount of safe transition \( qtdS \), actions that are triggered simultaneously \( RA \)
transitions \( t \leftarrow 0 \)
derivable products \( dP \leftarrow 0 \)
safe transitions \( st \leftarrow 0 \)
d\( P.add(P_0) \)
while \( t < qtd \) OU \( st < qtdS \) do
    for all product \( p \) in \( dP \) do
        for all \( ra \subseteq P \) of \( (RA) \) do
            \( p' \leftarrow ra \) applied over \( p \)
            if \( p' \) is valid then
                \( P.add(p') \)
                transition \( \{ p \rightarrow p', ra \} \) is reported as safe transition.
                \( t \leftarrow t + 1 \)
                \( tS \leftarrow tS + 1 \)
            else
                transition \( \{ p \rightarrow p', ra \} \) is reported as unsafe transition.
                \( t \leftarrow t + 1 \)
            end if
        end for
    end while

5 EVALUATION

To evaluate our tool, we perform it over a context-aware feature model from Mobiline, a SPL for Mobile and Context-Aware Visit Guide applications, proposed in (Marinho et al., 2010; Marinho et al., 2013). In a broader sense, it is an improved and indoor version of (Abowd et al., 1997). A fragment of system and context model are presented in Figs. 12 and 13.

Initially, Fixture checks the data model against the well-formedness properties defined in OCL. Once it is
recognized as a model that obeys all of these properties, the diagram is translated into propositional formulas following the process described in (Benavides et al., 2010). Then, every composition property is grouped by the conjunction operator. Finally, the restriction about boolean attributes described in Section 4.3 are also grouped by the conjunction operator. Having these three propositional formulas, we can evaluate them grouped and isolated. If we analyze them in a isolate way, we can obtain an answer to specific questions. For instance, if we analyze the propositional formula created in step two, we can determine if the composition properties are satisfiable or not. If we analyze the one created in step one, we determine if the feature diagram in itself yields any product.

This propositional formula covers features optionality and obligatoriness, the cardinality of OR and XOR feature groups, the composition properties and restrictions between attributes used in composition properties.

Fixture checks if the CAFM contains false optional and dead features. The answer to false optional is achieved asking the BDD solver to solve the propositional formula applying the boolean value false to each optional feature at a time. If the propositional formula is not satisfiable, so this optional feature is a false optional feature. To check dead features, we do the opposite process: we apply the boolean value true to each optional feature at a time and call the BDD solver. If the propositional formula is not satisfiable with those values means that the model has dead features.

Once the CAFM does not contain dead or false optional features, the user is requested to configure a product. Once the configuration process is over, Fixture checks whether this configuration is safe or not. This is achieved by transforming the configuration into a string of boolean values. These boolean values represent each of the variable created in the aforementioned processes. So, it is submitted to BDD solver that will inform the user about the safety of the product. If the current product configuration is said to be safe, we proceed to the simulation phase.

6 RELATED WORK

Benavides et al. (Benavides et al., 2005) stressed the use of automated method to reason over SPL. They used Constraint Satisfaction Problem and their work can be considered an evolution of (Van Deursen and Klint, 2004) and (Mannion, 2002). However, context-awareness is not even mentioned.

Fernandes et al. (Fernandes et al., 2011) proposed the UbiFex notation and UbiFex Simulation Process. The notation models variability with features but does not consider attributes for them which strongly ease the treatments. Moreover, in the simulation process, simultaneous context properties triggering are not considered. Finally, the simulation process is one-step process and because of this, it does not provided a vision of the system evolution.

Rincón et al. (Rincón et al., 2014) proposed an ontological rule-based approach to identify dead and false optional features, identify certain causes of these defects and explain these causes in natural language, but they did not regard context-awareness.

Hartmann and Trew (Hartmann and Trew, 2008) deal with context-awareness but in static way. Although it is focused on multiple product line and concerns the constrains between context and system model.

Wagelaar (Wagelaar, 2005) used ontologies to represent contextual informations. In this way, he was able to perform computations over the feature diagrams. However, part of these computations are limited by its own model representation.

Some gaps found in those works are: context-awareness (Benavides et al., 2005; Rincón et al., 2014), data models (Hartmann and Trew, 2008), separation between system and context models (Benavides et al., 2005; Rincón et al., 2014), detection of anomalies (Wagelaar, 2005; Hartmann and Trew, 2008), automatic reasoning hartmann and simulation (Fernandes et al., 2011). We claim that our work deals with these requirements even partially.

7 CONCLUSION AND FUTURE WORK

This work has provided a more rich data model to represent context and system parts of context-aware applications. Additionally, it contributes to expand the set of possible computations over context-aware applications.

Given the lack of automated tools, this work created a prototype tool that offers a graphical interface to make easier the development of context-aware applications. At the same time, we also focused on algorithms and formal methods (the well-formedness properties are specified in a OCL-like language) which clearly provides less errors in CAFM.

As future work, other verification and validation methods could be added to the tool for example to...
make possible the comparison and selection of different methods. In order to provide additional real simulations, the adaptations could be enhanced with probabilities of how many times they could occur. Furthermore, we intend to implement an enrichment semantic (Filho et al., 2012) process enabling to associate feature model with business assets.

ACKNOWLEDGEMENT

The authors would like to thank the researchers and students involved in the UbiStructure Project as well as CAPES and CNPq for the financial support.

REFERENCES


