Formal Specification of Software Product Lines: A Graph Transformation Based Approach

Khaled Khalfaoui
Department of Computer Science, University of Jijel, Jijel, Algeria
kh_khalfaoui@yahoo.fr

Allaoua Chaoui
MISC Laboratory, Department of Computer Science, University of Constantine, Algeria
a_chaoui2001@yahoo.com

Cherif Foudil
Department of Computer Science, University of Biskra, Biskra, Algeria
foud_cherif@yahoo.fr

Elhillali Kerkouche
Department of Computer Science, University of Jijel, Jijel, Algeria
elhillalik@yahoo.fr

Abstract—A Software Product Line is a set of software products that share a number of core properties but also differ in others. Differences and commonalities between products are typically described in terms of features. A Feature Diagram is a hierarchically structured model that defines the features and their dependencies, while a Featured Transition System is used concisely to model behaviour of each product. In this context, formal modeling and verification are critical for managing the inherent complexity of systems with a high degree of variability. This work presents a formal specification of Software Product Line models based on rewriting logic. We propose an automatic framework for translating featured transition system and feature diagram into an equivalent Maude specification. It is based on meta-modelling and graph transformation. The power of this translation resides in the fact that the proposed formalization preserves source models semantics. An illustrative example is presented. The approach allows various verification and analysis activities. The obtained results are significant.

***Index Terms*—Software Product Line, Featured Transition System, Feature Diagram, Specification, Verification, Rewriting Logic, Maude, Graph Transformation**

I. INTRODUCTION

Software product line engineering is an approach for developing families of software systems. A software product line (SPL) can be defined as a set of software products sharing a common set of features. The main advantage over traditional approaches is that all products can be developed and maintained together.

Usually, SPLs are modeled with Featured Transition Systems (FTS) [1] and Feature Diagrams (FDs) [2]. FTS allows concisely modeling the behaviour of each product in the SPL with a single parameterized model to be instantiated differently for each product. Whereas, FDs permit to model the variability of the SPL. The FD expresses the set of valid products. Since products are combinations of features, formal modeling and verification are critical for managing the inherent complexity of SPLs. With a high degree of variability, to manage the inherent complexity of the SPL models, formal modeling and verification are necessary. In this work, we are interested in rewriting logic (RL) [3].

The RL is a flexible and expressive semantic framework for the specification of systems behavior. It can be used for specifying a wide range of systems in various application fields. Several languages based on RL have been designed and implemented. Maude [4] is widely used. It is considered as one of languages in which many different kinds of systems can be naturally specified. In addition to its power of expression, Maude offers many possibilities of validation and verification. For validation, it supports simulation in a flexible way. For verification purpose, Maude supports model checking.

FTS and FD models can be expressed in RL. This formalization aims to use the formal analysis techniques developed for RL to analyze these models. In order to generate the Maude specification, we have proposed in [5] a manual approach. The main idea is that FTS transitions and its conditions firing are translated into conditional rewriting rules. The right hand side and the left hand side of each rule are FTS states. The condition is used to verify the presence and the priority of the feature.

For the systems verification purpose, graph transformation techniques are widely used. The aim is to transform system graphical models into their formal equivalent specifications supporting assessment and

doi:10.4304/jsw.7.11.2518-2532

analysis of characteristics. This task is performed by executing a graph grammar. A graph grammar [6] is composed of rules. Each one has a graph in their left and right hand sides (LHS and RHS). Rules are compared with an input graph called host graph. If a matching is found between the LHS of a rule and a subgraph in the host graph, then the rule can be applied and the matching subgraph of the host graph is replaced by the RHS of the rule. Furthermore, each rule may also have application conditions that must be satisfied, as well as actions to be performed when the rule is executed. A graph rewriting system iteratively applies rules of grammar in the host graph, until no rules are applicable.

In this paper we propose an approach for analyzing SPL models where we develop an automatic framework based on graph transformation to translate FTS and FD diagrams into an equivalent Maude specification. To this end, we have defined meta-models for FTS and FD formalisms. Then the meta-modelling tool AToM^3 [7] is used to automatically generate a visual modeling tool for each formalism according to its proposed meta-model. We have also proposed a graph grammar which performs the transformation of these models into semantically equivalent Maude specification. Our tool allows drawing FTS and FD models and transforming them automatically into their equivalent in RL. Once the equivalent Maude code is generated, the LTL model checker can be used. In order to perform the analysis using Maude's LTL model checker, we have to generate predicates and properties in Maude language.

This paper is organized as follows. Section 2 outlines some related works. In section 3, we recall some basic notions about FTS and FD diagrams. We give an overview of RL and Maude language in section 4. In section 5, we give an overview of graph transformation and the AToM^3 tool. In Section 6, we first introduce the proposed approach to specify FTS models in Maude language then define the meta-models and the graph grammars and finally present the verification process. In section 7, we illustrate our framework through an example. Finally, section 8 concludes the paper and gives some perspectives of this work.

II. Related Works

An SPL is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [8]. SPLs are used for the development of embedded and critical systems. Formal modelling and model checking of SPL behaviour is thus vital for quality assurance.

Over the past few years, several modelling and analysis techniques have been published. Larsen et al. [9] propose modal I/O automata to model variability in component interfaces and discuss compatibility between these interfaces. In a similar effort, Fischbein et al. [10] propose modal transition systems (MTS) to model SPLs and examine the notions of behavioural conformance in MTS that are suitable for SPLE. Fantechi and Gnesi [11] extended their approach by introducing explicit variability operators into MTS. In [12] Asirelli et al. apply deontic logic to express both static and behavioral aspects of product families.

These approaches even though they are formal, do not provide mechanisms for the verification of temporal properties. To correct this problem, Li et al. [13] propose compositional approach for CTL model checking of features. A feature automaton can be attached to two precisely defined interface states of the base system. In [14] Lauenroth et al. propose to use automata labelled with features and give an algorithm for CTL model checking over automata. The algorithms they propose do not attempt to explore the state space in an efficient manner. Classen et al. in [1] has proposed an algorithm that can treat this problem more efficiently. They had addressed the model checking problem for SPLs and linear temporal logic (LTL) by introducing FTSs, a mathematical formalism to express the behaviour of all products of the SPL in one model.

Nowadays, meta-modelling and graph grammars are widely used for modelling and analysis of complex systems in the area of software engineering. There are many researches working on the topic related to model-driven engineering (MDE). In [15], it has been proposed a transformation between Statecharts and Petri Nets. In [16] the authors have proposed a tool that formally transforms dynamic behaviours of systems expressed using UML Statechart and collaboration diagrams into their equivalent colored petri nets (CPN) models. To make the analysis complete and robust, they have used the obtained CPN models to generate automatically their equivalent description in the input language of the Petri net analyzer INA. In [17], for analysis and verification, UML activity diagrams have been translated into an equivalent Communicating Sequential Processes (CSP) specification using an approach based on graph transformations.

The current paper presents a first attempt towards a formal specification of Software Product Line models based on graph transformation and RL. The SPL products are modeled by means of FTS and FD diagrams.

III. Variabily Modeling

For SPLs modeling, several techniques have been proposed. In this work, we are interested in FTS and FD diagrams. FD is used to express the structural view of the SPL. On the other side, FTS is used to describe the combined behaviour of the entire system family.

A. Feature Diagram

FD, feature diagram, is a graphical representation which shows a hierarchically structured set of features of the product line. Features are represented as nodes and relationships between features as links. Possible relationships between features are usually categorized as "And" (all subfeatures must be included), "Or" (one or more subfeatures can be included), "Alternative" (only one subfeature can be included), "Mandatory" (required feature), and "Optional" (potential feature). A feature

diagram is typically represented as a tree where primitive features are leaves and compound features are interior nodes.

In Software Product Line Engineering (SPLE), systems are developed in families and differences between members of a family are generally represented by features. A set of features can be seen as the specification of a product. An FD is a concise representation for the valid products of an SPL. As an example, consider the FD of a vending machine SPL (inspired from [1]) presented in Fig.1.

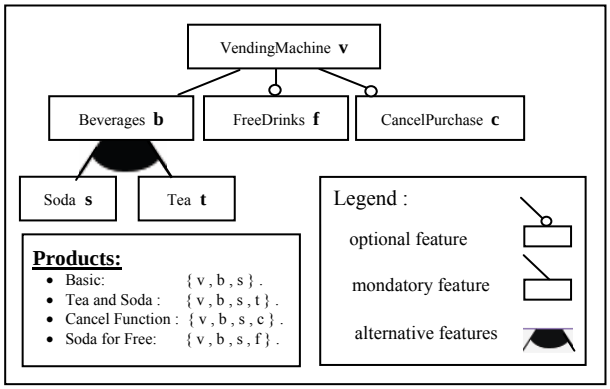


Figure 1. Feature Diagram

Consider four variants of this machine:

$$\{v, b, s\}, \{v, b, s, t\}, \{v, b, s, c\}, \{v, b, s, f\}$$
$$P_1 \qquad P_2 \qquad P_3 \qquad P_4$$

The first variant P_1 sells soda. The second P_2 sells soda or tea. The third one P_3 lets the buyer cancel her purchase after entering a coin. The last P_4 offers free drinks.

B. Featured Transition System

FTS, featured transition system, is a formalism designed to describe the combined behaviour of a whole system family. FTS is transition system (TS) in which transitions are labelled with features of an FD in addition to being labelled with actions [1]. A transition is part of a product if and only if its feature is part of the product. In FTSs there can be priorities between transitions to model the case in which a feature removes, rather than adds, transitions. The FTS for the vending machine example is given in Fig.2.

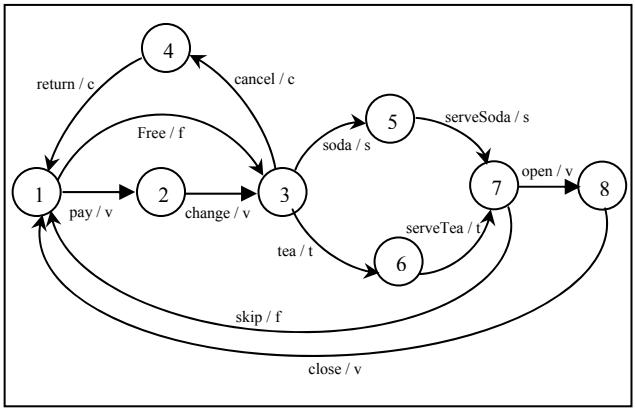


Figure 2. Featured Transition System

Intuitively, the FTS captures impact of all features in a single diagram.

Priority

A transition $s \rightarrow s_1$ labelled with f_1 has priority over $s \rightarrow s_2$ labelled with f_2, written: $s \rightarrow s_1 > s \rightarrow s_2$, iff f_2 is an ancestor of f_1 in FD.

A common modeling pattern is that the behavior of a child feature overrides the behavior of its parents. In order to obtain the behavior of a particular product, it is necessary to project the FTS on the set of features corresponding to a valid product. This transformation is entirely syntactical and consists in removing (i) all transitions linked to features that are not in this product, and (ii) all transitions that are overridden by higher priority transitions. The result of the projection is an ordinary TS.

Diagrams (a), (b), (c) and (d) of Figure 3 represent respectively the behavior of products P_1, P_2, P_3 and P_4.

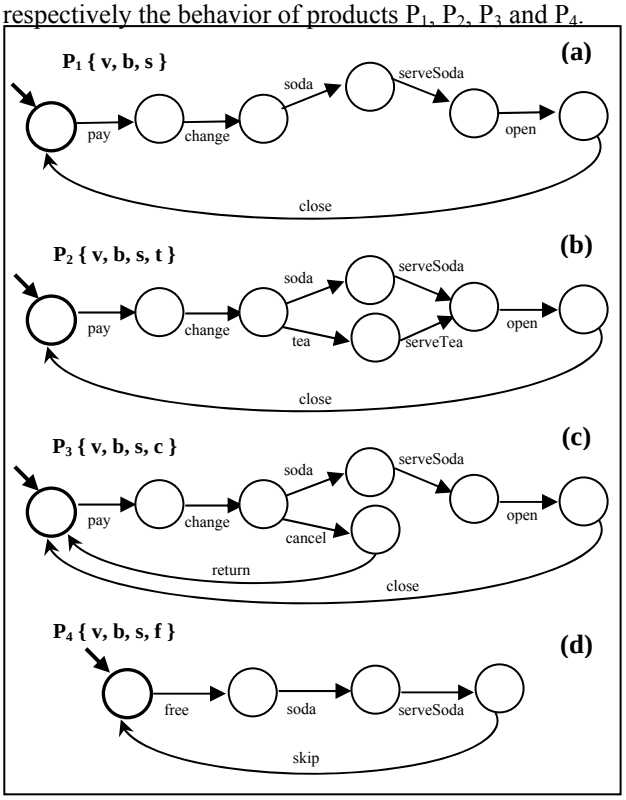


Figure 3. SPL products

IV. REWRITING LOGIC AND MAUDE

RL is a computational logic proposed by Meseguer as a unified logic for concurrency [3], which builds upon equational logic by extending it with rewrite rules. In RL, each concurrent system can be specified easily by a rewriting theory. A rewrite theory is defined as a 4-tuple (Σ, E, L, R). The signature (Σ, E) is an equational theory, L is a set of labels, and R is a set of possibly conditional labeled rewrite rules that are applied modulo the equations E. An important consequence of the RL definition is that the rewrite theory can be viewed as an executable specification of the concurrent system that it formalizes. The state is represented by an algebraic term,

the transition becomes a rewriting rule and the distributed structure is expressed as an algebraic structure. For more information on the subject see [3].

Maude is a specification and programming language based on RL [18]. It integrates an equational style of functional programming with RL computation. Maude's implementation has been designed with the explicit goals of supporting executable specification and formal methods applications. Because of its efficient rewriting engine, it is considered as an excellent tool. It is simple, expressive and efficient. Three types of modules are defined in Maude: The functional modules, the system modules and the object oriented modules. In this work, we will use only functional and system modules.

- **Functional Modules:** Functional modules define data types and operations on them by means of equational theories. By using equations like simplification rules, each expression could be evaluated to its reduced form called the canonical form. The result is the same regardless of the order of application of the equations. This ensures that the initial algebra and the canonical term algebra of the functional module are isomorphic, and therefore that the module's mathematical and operational semantics coincide [18]. From a programming point of view, a functional module is an equational-style functional program with user definable syntax in which a number of sorts, their elements, and functions on those sorts are defined.
- **System Module:** The system module defines the dynamic behavior of a system. It specifies a rewrite theory. A rewrite theory has sorts, kinds, and operators, and can have three types of statements: equations, memberships, and rules, all of which can be conditional [18]. A rewriting rule specifies a local concurrent transition which can proceed in a system. The execution of such transition, specified by the rule, can take place when the left part of a rule matches to a portion of the global state of the system and the condition of the rule is valid. This type of module augments the functional modules by the introduction of rewriting rules. From a programming point of view, a system module is a declarative-style concurrent program with user definable syntax.

In addition, Maude also integrates a model checker. Model-checking is an automatic method for deciding if specification model, expressed as a concurrent transition system, satisfies a set of properties. Model checking supported by the Maude's platform uses LTL [19] logic for its simplicity and the well-defined procedures of decision which it offers. The Maude LTL model checker is efficient (for more details, see [20]).

V. GRAPH TRANSFORMATION

A. Graph Grammar

Graphs are well-known and frequently used to represent complex objects and diagrams [6]. Rules have proved to be extremely useful for describing computations by local transformation. Graph transformation (also known as graph rewriting) combines the advantages of both into an individual computational paradigm.

A graph transformation rule (Fig.4) is a special pair of pattern graphs where the instance defined by the left hand side (LHS) is substituted with the instance defined by the right hand side (RHS) when applying such rule. Rules are local in a sense that they handle only a small amount of model elements, and therefore the designer does not need to concentrate on the entire transformation problem.

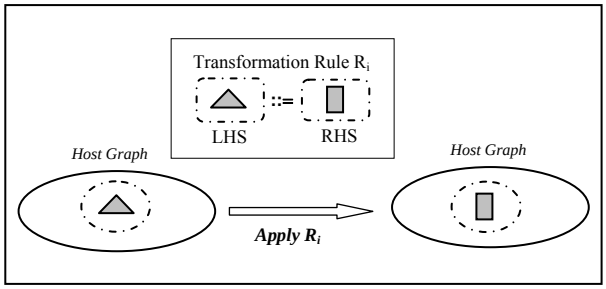


Figure 4. Graph transformation rule

Graph transformation rules are usually called graph grammars. These are a generalization of Chomsky grammars for graphs [21]. In the rewriting process, rules are evaluated against an input graph, called the host graph. If a matching is found between the LHS of a rule and a subgraph of the host graph, then the rule can be applied. When a rule is applied, the matching subgraph of the host graph is replaced by the RHS of the rule. Rules can have applicability conditions, as well as actions to be performed when the rule is applied. Some graph rewriting systems have control mechanisms to determine the order in which rules are checked. Generally, rules are ordered according to a priority assigned by the user and are checked from the higher priority to the lower priority. After a rule matching and subsequent application, the graph rewriting system starts again the search. The graph grammar execution ends when no more matching rules are found.

The use of small subgraphs on the LHS of graph grammar rules, as well as using attributes, can greatly reduce the search space. This is the case with the vast majority of the used formalisms in this field of research. There are three kinds of transformations. The first is model execution (defining the operational semantics of the formalism). The second is model transformation into formalism. A special case of this is when the target formalism is textual. The third one is model optimization, for example reducing its complexity.

Graph grammars are a natural, formal, visual, declarative and high-level representation of the computation.

B. Meta-modelling

In the field of graph transformation, the meta-modelling technique is widely used to describe the different kinds of formalisms needed in the specification and design of systems. To define a meta-model, we have to provide two syntaxes. On one hand, the abstract formal

syntax to denote the formalism's entities, their attributes, their relationships and the constraints. To do this, we usually use a graphical modelling notations such as UML class diagrams or Entity-Relationship Diagrams. On the other hand, the concrete graphical syntax to define graphical appearance of these entities and relationships. Once the meta-model is defined, meta-modelling environments are able to automatically produce a visual interactive tool for the defined formalism. The advantage of this technique is that the generated tool accepts only syntactically correct models according to the formalism definition. For more details see [22].

C. AToM^3

AToM^3 [7] is a visual tool for multi-formalism modeling and meta-modelling. Being implemented in Python [23], it is able to run without any change on all platforms for which an interpreter for Python is available. The AToM^3 meta-layer allows a high-level description of models using the Entity-Relationship (ER) formalism extended with the ability to express constraints. Based on these descriptions, AToM^3 can automatically generate tools to visually manipulate (create and edit) models in the formalisms of interest [22].

The AToM^3 graph rewriting system uses graph grammars to visually guide the procedure of model transformation. Model transformation refers to the automatic process of converting, translating, or modifying a model of a given formalism into another model that might or might not be in the same formalism.

In AToM^3, rules are ordered according to a user-assigned priority, and are checked from higher to lower priority. In the LHS of rules, the attributes of the nodes must be provided with attribute values which will be compared with the nodes attributes of the host graph during the matching process. These attributes can be set to *<ANY>* or have specific values. In order to specify the mapping between LHS and RHS, nodes in both LHS and RHS are identified by means of labels (numbers). If a node label appears in the LHS of a rule, but not in the RHS, then the node is deleted when the rule is applied. Conversely, if a node label appears in the RHS but not in the LHS, then the node is created when the rule is applied. Finally, if a node label appears both in the LHS and in the RHS of a rule, the node is not deleted. If a node is created or maintained by a rule, we must specify in the RHS the attributes' values after the rule application. In AToM^3 there are several possibilities. If the node label is already present in the LHS, the attribute value can be copied (*<Copied>).* We also have the option to assign it a specific value by giving the Python code to calculate this value (*<Specified>*), possibly using the value of other attributes. In addition, AToM^3 allows the use of global attributes available in all of the graph grammar rules as well as constraints.

The combined use of meta-modelling and graph grammars taken in AToM^3 allow users not only to benefit from the advantages of both (meta-modelling and graph grammars) but also to model with multi-paradigm Modeling [22]. The AToM^3 has been proven to be a powerful tool.

VI. Our approach

In this section, we present our technique used for the specification and verification of the SPL models.

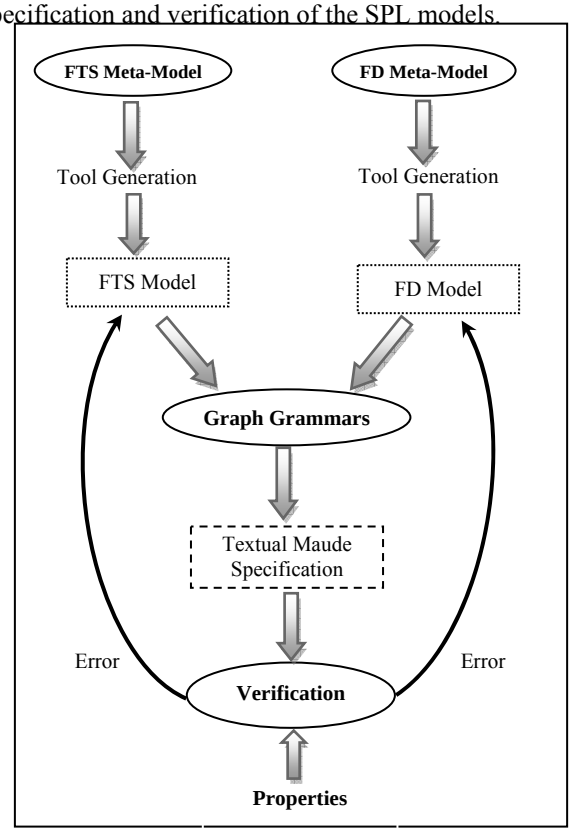


Figure 5. The general outline of the proposed approach

Our approach consists of a process with three steps:
The first step consists of meta-modelling FTS and FD formalisms and to generate automatically a visual modeling tool for each of them using AToM^3. The second step is to define the graph transformation grammars. The last one is the analysis of the generated Maude specification.
Before describing in detail the previous steps, it is preferable to begin by introducing the idea behind the specification of FTS models in Maude language.

A. Formalization

On one hand, to manipulate features, we define a functional module *Feature_FunctMod* that contains the declaration of a new type called *Feature* and the definition of operations used for manipulating sets of features, as well as equations implementing these operations. On the other hand, for specification and treatments of FTS states, we define a second functional module *FTS_FunctMod*. Classical TS states are represented as constants of a new sort *TsState*. We define the operation "<_;_;_>" to specify the current FTS state. The first parameter of this operation is a constant of the sort *TsState*. The second one is the set of all features specific to the considered product. The last one is Boolean indicating whether or not this state is final. This

latter is introduced to stop the evolution of FTS once a final state reached.

FTS transitions firing and its conditions are translated into conditional rules. In our approach, a rewrite rule has a structure of the form:

crl < Trans > : FtsState_From → FtsState_To if Pres&Prior .

where:

- FtsState_From and FtsState_To: are respectively the left and the right hand sides of the rule. These are two FTS states.
- Trans: is the transition name.
- Pres&Prior: is a Boolean term that specifies the condition of the transition Trans.

There are two possible configurations:

- Configuration$_1$: when there is only one output transition from a state (Fig.6).

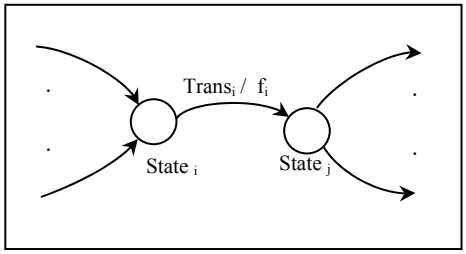


Figure 6. Configuration$_1$

This transition is enabled, when the feature f_i is in the set of selected features of the SPL considered product. This transition is specified in Maude language as follows:

crl < Tran$_i$ > : < State$_i$, ListSelectFeats ; false > -> < State$_j$; ListSelectFeats ; flag > if IsIn (f$_i$, ListSelectFeats).

where:

- ListSelectFeats: is the set of all features specific to the considered product.
- flag: is the Boolean indicating whether the State$_j$ is a final state or not.
- IsIn (f_i , ListSelectFeats): Boolean function indicating whether the feature f_i is in a ListSelectFeats or not.

- Configuration$_2$: when there is more than one output transition from a state (Fig.7).

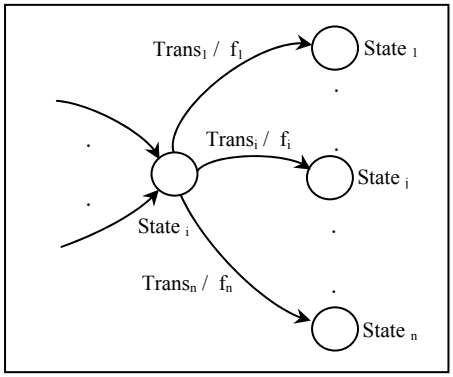


Figure 7. Configuration$_2$

Here, the transition Trans$_i$ in Fig.6 is enabled when two conditions are simultaneously satisfied:

- The feature f_i is in ListSelectFeats.
- f_i has a higher priority over all the features f_k ($1 \leq k \leq n$ and $k \neq i$) if f_k is in ListSelectFeats.

As shown in section II, the behaviour of a child feature in FD overrides the behaviour of its parents. For this reason, we propose to use a set containing all descendants of the feature f_i. This transition is specified in Maude language as follows:

crl < Trans$_i$ > : < State$_i$; ListSelectFeats ; true > -> < State$_j$; ListSelectFeats ; flag >
if IsIn (f$_i$, ListSelectFeats) and
(not IsIn (f$_k$, ListSelectFeats) or
(IsIn (f$_k$, ListSelectFeats) and not IsIn (f$_k$, SetOfDesc_f$_i$))).

where:

- SetOfDesc_f_i: the set of descendants of the feature f_i.

B. *Automatic Translation and Verification*

In the following, we present the three steps of our approach.

Step1: Meta-modelling

The meta-formalism used is the Entity-Relationship diagram. To implement the previous translation, we propose to add some additional attributes in our meta-models.

1- FD meta-model:

FD models consist of nodes and links between these nodes. We propose a meta-model called *FD_MetaModel* with an entity *FD-Feature* representing features and a relationship *FD-HasChild* for links as shown in Fig.8.

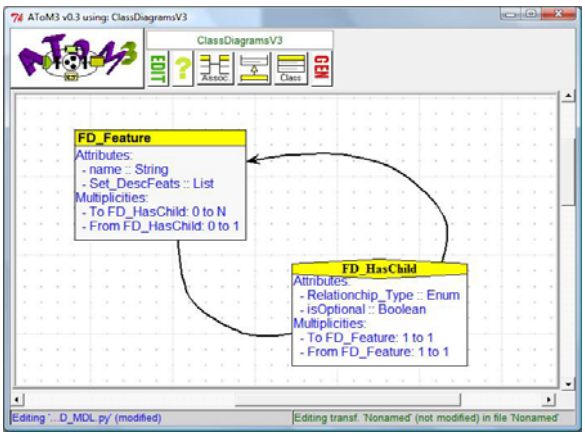


Figure 8. FD meta-model

FD-Feature Entity: It has two attributes: its name and the set of all its descendents called *Set_DescFeats*. Like shown in the previous section, this latter will be used in the specification of the FTS transitions.

FD-HasChild Relationship: It represents the family relationship between two features. The destination feature is a child of the source feature. No attribute is used.

2- FTS meta-model:

An FTS model (Fig.2) consists of states and transitions. So, we propose a meta-model called *FTS-MetaModel* with only one entity *TS-state* describing states, and one relationship *FTS-Transition* describing transitions (Fig.9).

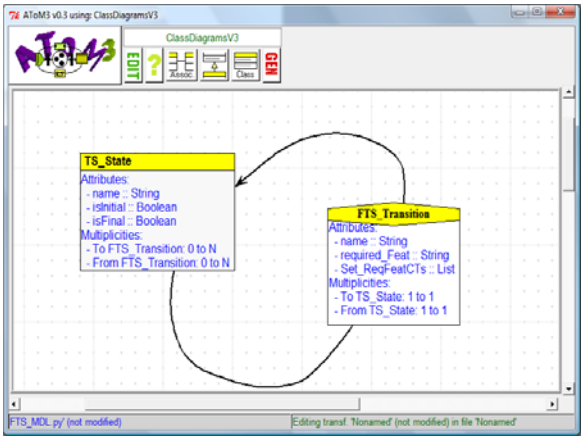


Figure 9. FTS meta-model

TS-State Entity: Each state has three attributes: its identifier (*name*) and two Booleans indicating respectively whether this state is an initial state (*Initial_State*) or is a final state (*Final_State*).

FTS-Transition Relationship: It represents the transition from a source state to a destination state. Each transition has three attributes. The first is its identifier (*name*). The second is the required feature (*required_Feat*). The third is a set of features (*Set_ReqFeatCTs*). This latter is proposed to contain all the features required in the transitions leaving the same state as this transition. It will be used to translate transitions of the second configuration.

To fully define our meta-models, we have also specified the graphical appearance of each entity of the FTS and FD formalisms according to its appropriate notation. Given our meta-models, we use AToM^3 tool to generate the visual modelling environments for these formalisms (FTS and FD). More precisely, AToM^3 generates, for each formalism, a palette of buttons allowing the user to manipulate the entities defined in the meta-model. As AToM^3 is a visual tool for multi-formalism modelling [7], we employ a user interface with the two generated tools at the same time (see Fig.19).

***Step2:* Defining the graph grammars**

In order to make the transformation easier, we propose to use three complementary graph grammars (see Fig.10). Briefly, the first graph grammar generates the functional module *Feature_FunctMod* and produces the set *Set_DescFeats* for each feature in the FD diagram. The second graph grammar generates the functional module *FTS_FunctMod* and works out for each feature of the FTS model the attribute *Set_ReqFeatCTs*. The last one generates the system module containing the rewriting rules called *FTS-SysMod*.

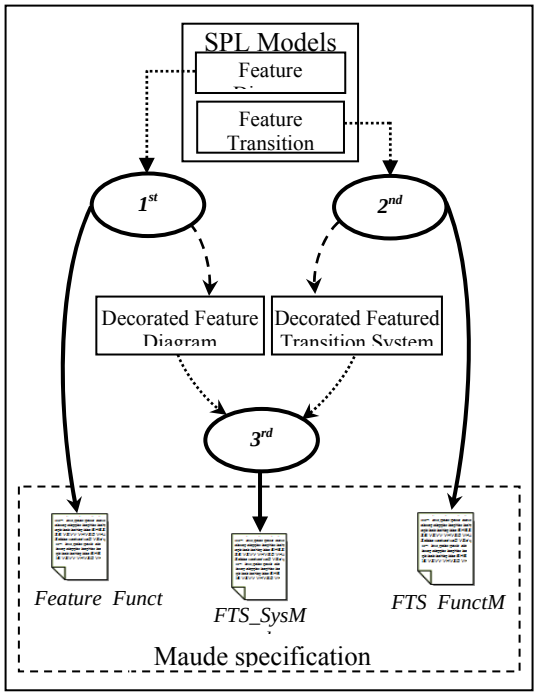


Figure 10. Transformation process

The three graph grammars are composed of transformation rules. In addition to the LHS and RHS, each rule is provided with:

- A priority.
- Conditions which must be satisfied to apply this rule.

In the execution of each grammar, the rewriting system iteratively applies matching rules in this grammar to the host graph, until no more rules are applicable. Rules are tried in ascending order.

1st GG: Gen_FeatFunctMod.

The *Feature_FunctMod* module consists of three parts:

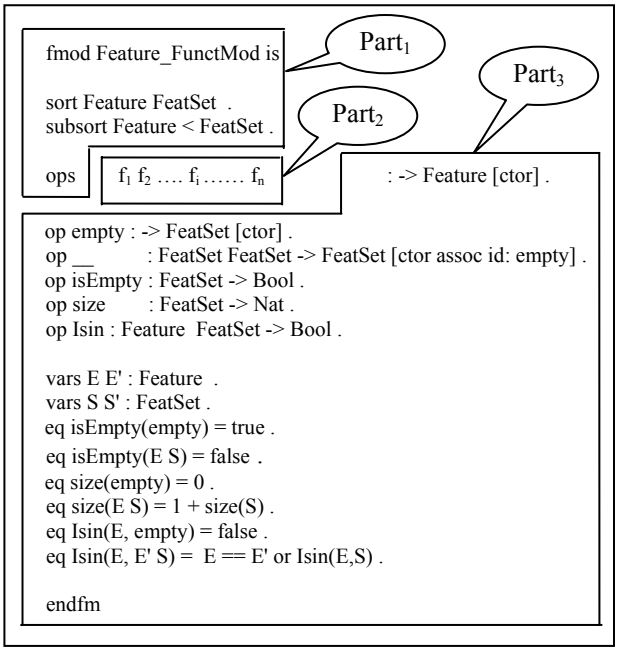


Figure 11. The functional module *Feature_FunctMod*

Gen_FeatFunctMod graph grammar has an initial action which creates and opens a textual file to generate the functional module *Feature_FunctMod*. At first, it generates the first part ($Part_1$). Then, it decorates all nodes (features) in the FD model with an auxiliary attribute called *Visited*. This later is used to determine whether the node has been previously treated or not. Treatment begins with the leaves. Each time the rewriting system locates a leaf not yet visited, it adds his feature to the attribute *Set_DescFeats* of its parent, and this leaf will be marked as visited. Then, we move on to intermediate nodes. In this case, the rewriting system only deals the nodes whose children were all treated. To do this, we use another temporary variable *Count_PrChild* in order to count the number of treated children. Similarly, for each node processed, we add the feature and its descendants to the attribute *Set_DescFeats* of its parent, and it will be marked as visited. At last, we mark the root node as visited. At the same time, for each visited node the feature is added to the text file to generate the second part ($Part_2$) of the functional module *Feature_FunctMod*. The final action is used to generate the third part of the functional module and to delete all used temporary attributes.

The proposed graph grammar (Fig.12) is composed of three rules.

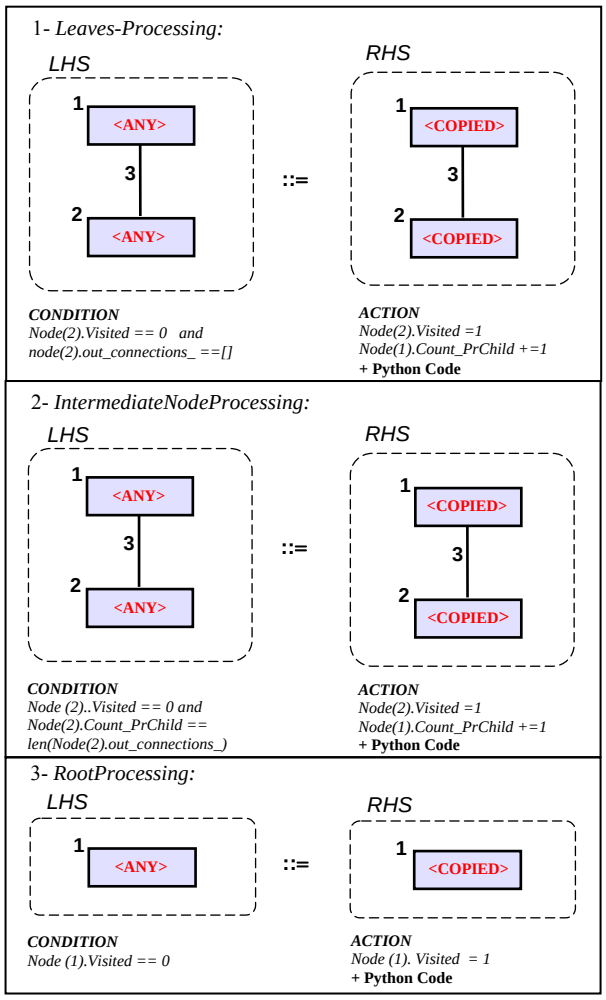


Figure 12. *Gen_FeatFunctMod* graph grammar

These rules are described as follows:

Leaves-Processing (Priority 1): is applied to process all leaves nodes. Each time, it locates a leaf node that has not been previously visited to add its name attribute in the *Set_DescFeats* attribute of its parent. Its *Set_DescFeats* attribute is set empty. To process another node, this leaf node will be marked as visited.

IntermediateNodeProcessing (Priority 2): is applied to process nodes which are located between the root and the leaves. At each iteration, it locates a node not yet visited and whose all children have been visited. Its name attribute and all its descendants will be added to its parent *Set_DescFeats* attribute. To avoid this process once again, it will be marked as visited.

RootProcessing(Priority 3): marks the root node as visited. Its *Set_DescFeats* attribute is already calculated by the second rule.

2nd GG: Gen_FTSFunctMod.

The functional module *FTS_FunctMod* consists of three parts (see Fig.13).

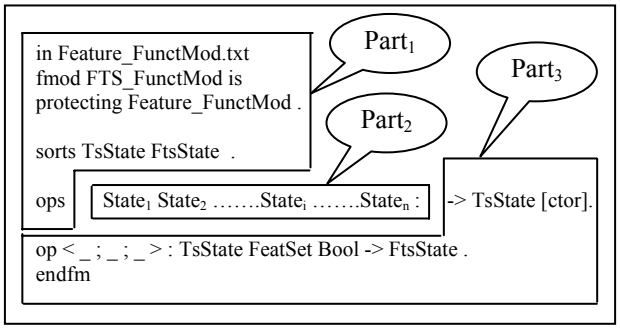


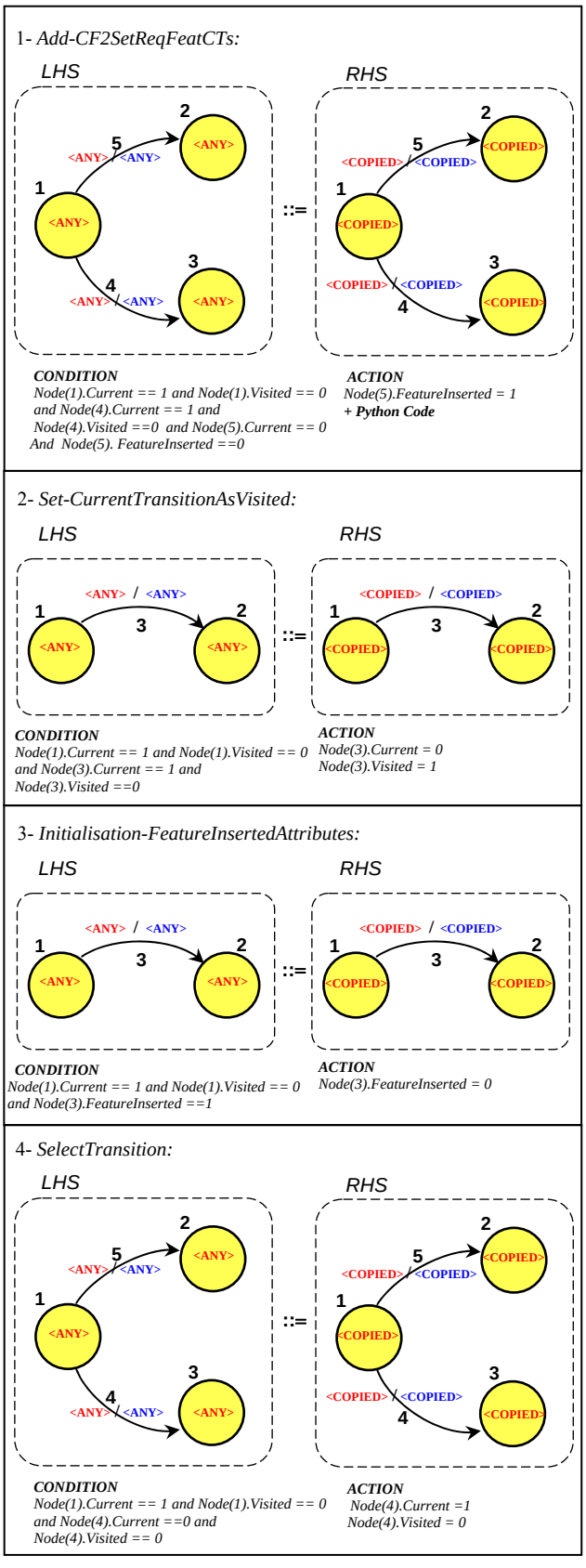
Figure 13. The functional module *FTS_FunctMod*

This graph grammar has an initial action that creates and opens a textual file to produce the functional module *FTS_FunctMod*. At the beginning, in its initial action, it generates the first part ($Part_1$) and decorates the entities in the FTS model with the used auxiliary attributes.

The idea behind the transformation is to pass through the FTS states one by one. First, the treated state is added to the textual file to generate the second part ($Part_2$). Then, we will treat all outgoing transitions. For each transition, we produce the attribute *Set_ReqFeatCTs* passing through the concurrent transitions one by one. To do this, we use two attributes for states, *Current* and *Visited*. The *Current* attribute is used to identify the state in the FTS model for which we will treat all output transitions, whereas the *Visited* attribute is used to indicate whether this state has already been treated or not. For the treatment of the outgoing transitions of the current state, we use three attributes *Current*, *Visited* and *FeatureInserted*. The *Visited* attribute is used to indicate whether the attribute *Set_ReqFeatCTs* of this transition has been produced or not. The *Current* attribute is used to indicate whether it is the transition for which we produce the attribute *Set_ReqFeatCTs*. The *FeatureInserted* attribute is used to indicate whether the feature required in this transition has been previously added to the set *Set_ReqFeatCTs* of the current transition

or not. At last, the final action generates the third part of the file and destroys all the used temporary attributes.

To carry out this process, we propose a graph grammar composed of seven rules.



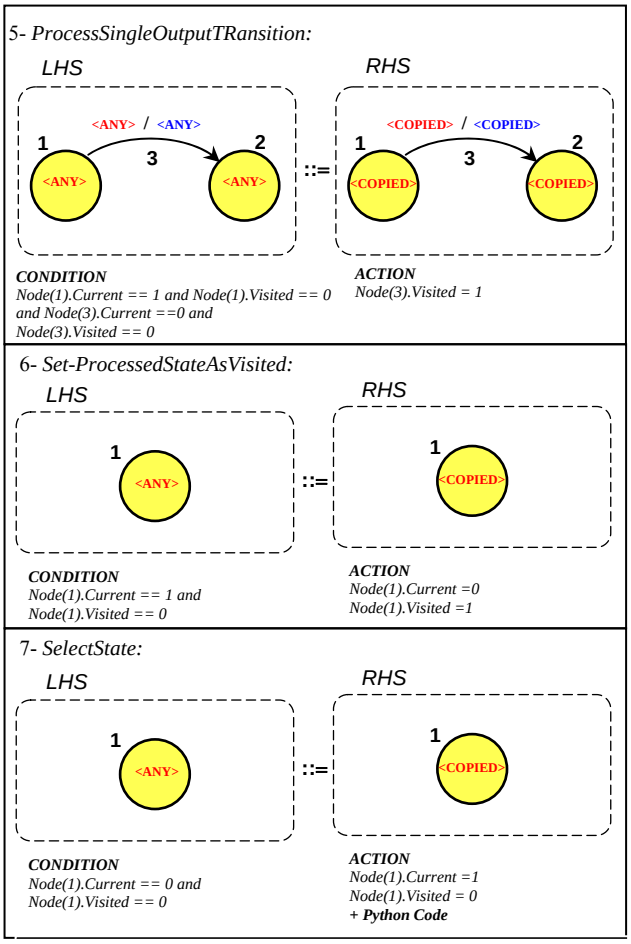


Figure 14. *Gen_FTSFunctMod* graph grammar

Add-CF2SetReqFeatCTs (priority 1): is applied to locate an output transition from current state that has not been previously visited in order to add its required feature to the *Set_ReqFeatCTs* attribute of the transition in process.

Set-CurrentTransitionAsVisited (priority 2): Once all features required in concurrent transitions are inserted in *Set_ReqFeatCTs* attribute of the current transition, this rule marks this latter as visited.

Initialisation-FeatureInsertedAttributes(priority 3): This rule is applied to initialise the *FeatureInserted* attribute of all output transitions of the current state to process another transition which is not yet treated.

SelectTransition: (priority 4): is applied to select a transition that has not been previously processed and which has the current state as source state to produce its *Set_ReqFeatCTs* attribute. This rule treats the case where there is more than one output transition from the current state ($Configuration_2$). Subsequently, rules *N°1*, *N°2* and *N°3* will be triggered.

ProcessSingleOutputTRansition(priority 5): This rule treats the case where the current transition is the single output transition from the current state ($Configuration_1$). It marks this transition as visited and its attribute *Set_ReqFeatCTs* is set empty. In this case, the fourth first rules are not applied.

Set-ProcessedStateAsVisited (priority 6): This rule, once all the output transition(s) of the current state have been

processed, is applied in order to update temporary attributes of the processed state and set it as visited.

SelectState(priority 7): is applied to select a state from FTS model that has not been previously visited to produce the *Set_ReqFeatCTs* attribute of all its output transitions. The name of the selected state is added to the text file to generate the second part ($Part_2$) of the functional module *FTS_FunctMod*.

3rd GG : Gen_SystemMod.

FTS_SysMod module (Fig.15) consists of two parts. The first is standard for all SPL families, while the second contains the rewriting rules specific to the studied SPL. Each transition of the FTS model and its firing conditions is translated into a conditional rule. The FTS transitions will be treated one after another. The rewriting system looks for a transition that is not already translated and treats it, then passes it to another.

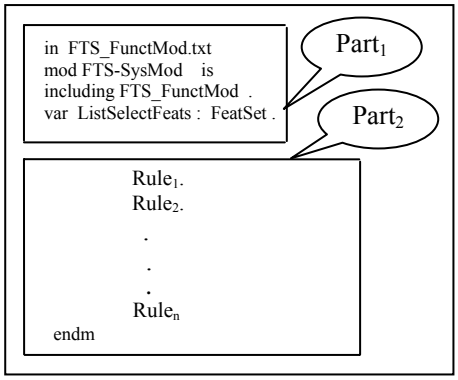


Figure 15. The system module *FTS_SysMod*

As proposed in our specification, each rewriting rule has a structure of the form:

crl < Trans > : FtsState_From → FtsState_To if Pres&Prior .

To facilitate the publication of a rule, we propose to divide it into two segments as shown in Fig.16.

The proposed *Gen_SystemMod* graph grammar has an initial action that creates and opens a textual file. Then, it generates the first part of the *FTS_SysMod* module and decorates all the states and transitions elements in the FTS model with temporary attributes to be used in the conditions specified in the rules. To generate the second part ($Part_2$), we propose to visit and to generate the specification of all FTS transitions one by one. For each transition, the first segment is generated in the same way. It contains the FTS_StateFrom and the FTS_StateTo. To generate the second segment, there are two configurations. If there are no other output transitions from its source state, we have to generate just the code verifying the presence of the required feature. Whereas, if there are more that one output transitions from its source state, we have to add the code specifying the priority conditions. The feature required in the considered transition must have higher priority over all features required in the concurrent transitions which are already in the *Set_ReqFeatCTs* attribute. This attribute has been calculated by the second graph grammar. To do this, we propose to use the following temporary attributes in transitions elements: *Visited* to indicate whether the code for this transition has been yet generated or not. *Current* to identify the transition in the FTS model whose code has to be generated. We add a third attribute called *Step* to generate the firing conditions in two stages. We first edit the presence condition (*Step=1*). Then, if it is the second configuration, we edit the priority conditions (*Step=2*). This process will be repeated for all other transitions in the FTS model which are not yet visited. The final action deletes all used auxiliary attributes.

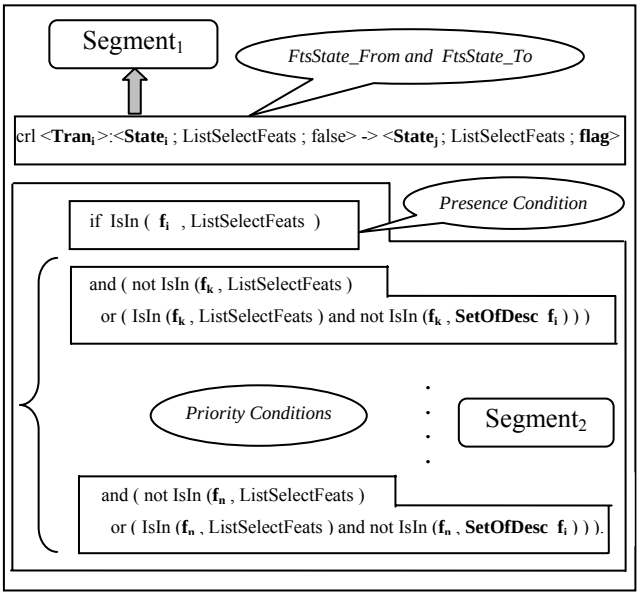


Figure 16. Specification of an FTS transition in Maude

To generate the module system, we propose a graph grammar with four rules. These rules are shown in Fig.17 and described as follows:

GenLHSandRHS-Rule (priority 1): is applied to generate in the text file the left hand side and the right hand side of the rewriting rule (segment1). This part contains the name of the current transition ($Tran_i$), the source state ($State_i$), the destination state ($State_j$) and the Boolean flag. If the destination state's attribute final is true, this Boolean is set true. Otherwise, it is set false.

GenPresAndPriority-Conds (priority 2): is applied to generate firing conditions. First, it generates in the text file the appropriate Maude code for checking the presence of the required feature (f_i) in the set of selected features for the considered product (*ListSelectFeats*). Then, in the case of the second configuration (*Set_ReqFeatCTs* is not empty), it generates the appropriate Maude code checking the priority conditions. To do this, the rule runs through all elements of *Set_ReqFeatCTs* attribute, and for each one it checks whether it is one of the descendants of the required feature or not. For this, we use the attribute *Set_DescFeats* of the node feature in FD diagram that has the same name as the required feature in the current transition (f_i).

SetCurrentTransitionAsVisited (priority 3): it locates the current transition whose processing has been terminated and marks it as Visited.

Select-Transition (priority 4): is applied to locate an FTS transition that has not been previously processed to translate it into a rewriting rule and marks it as current.

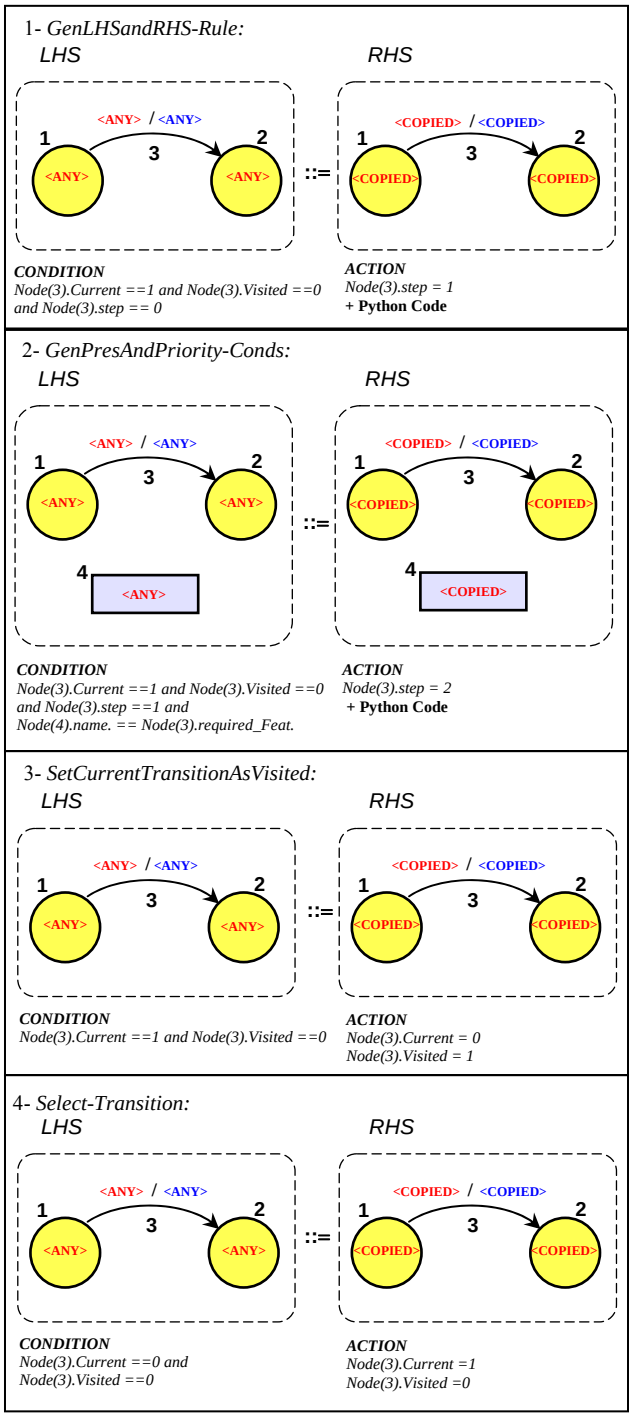


Figure 17. *Gen_SystemMod* Graph Grammar

***Step3:* Verification and analysis**

After generating automatically the SPL Maude specification, we pass to the verification and analysis. To check the behaviour of a given product, the user has to:

- Specify this product by giving an initial state. This latter is an FTS state that contains the set of its specific features.
- Describe manually the property to be verified with an LTL formula.

Then, the model-check function can be called. Maude model checker verifies automatically if the LTL formula is valid in this state or in the set of all accessible states from the initial state. If the formula is not valid, a counterexample is displayed (Fig.18). In this case, the FTS and FD models present errors. They must be corrected.

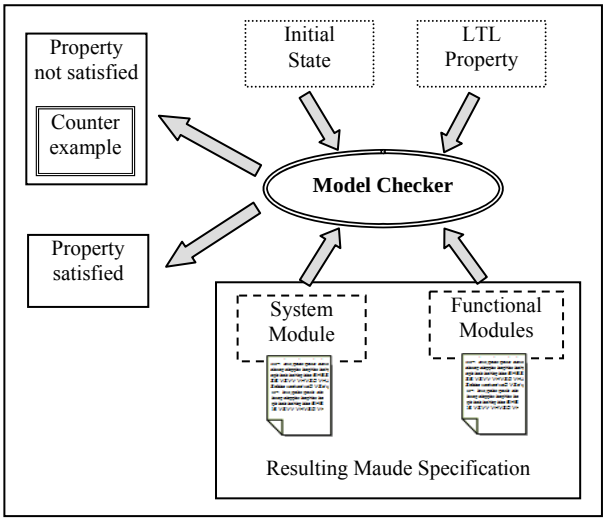


Figure 18. Model Checking

It should be noted that:

- The proposed meta-models and graph grammars are standards and applicable for all SPLs, whereas the LTL formulas must be redefined to each family of products studied according on the property to check.
- For the three graph grammars, we are concerned by calculating attributes value or code generation. So, none of the proposed rules changes the input models.
- The resulting Maude specification expresses the behaviour of all products of the SPL.

VII. ILLUSTRATIVE EXAMPLE

To illustrate our framework, let us consider the vending machine example which was seen previously (Section 2). As input, we have to create FD and FTS models using the generated user interface as shown in Fig.19. The toolkit provided allows manipulating all entities of the two formalisms.

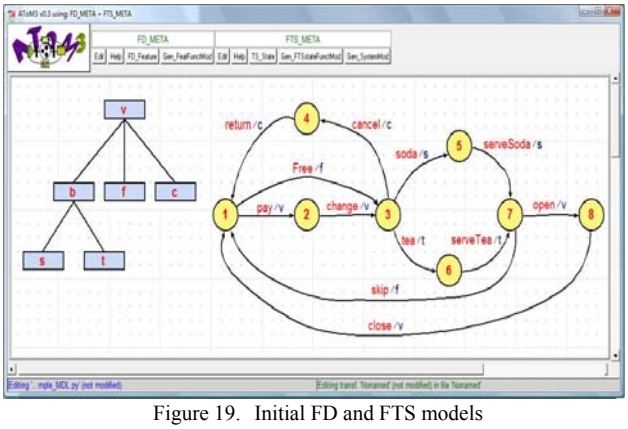


Figure 19. Initial FD and FTS models

In order to translate these models into the equivalent Maude specification, we have to apply the three previously defined graph grammars. First, by executing *Gen_FeatFunctMod* graph grammar on the FD model, we obtain the functional module *Feature_FunctMod* (Fig.20) and a decorated FD model for which each feature is enriched with the set of its descendants (Fig.21).

```
fmod Feature_FunctMod is
sort Feature FeatSet .
subsort Feature < FeatSet .

ops v b s c t f : -> Feature [ctor] .
op empty : -> FeatSet [ctor] .
op __ : FeatSet FeatSet -> FeatSet [ctor assoc id: empty] .
op isEmpty : FeatSet -> Bool .
op size : FeatSet -> Nat .
op Isin : Feature FeatSet -> Bool .
vars E E' : Feature .
vars S S' : FeatSet .
eq isEmpty(empty) = true .
eq isEmpty(E S) = false .
eq size(empty) = 0 .
eq size(E S) = 1 + size(S) .
eq Isin(E, empty) = false .
eq Isin(E, E' S) = E == E' or Isin(E,S) .

endfm
```

Figure 20. The functional module *Feature_FunctMod*

For example, Fig.21 shows that the attribute *Set_DescFeats* of the feature b contains the features s and t which are exactly the descendants of the feature b.

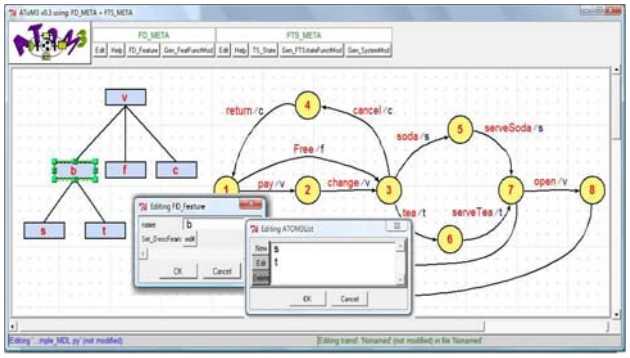


Figure 21. Decorated FD model

Then, to generate the functional module *FTS_FunctMod* (Fig.22) and to extend the FTS model we have to execute the *Gen_FTSFunctMod* graph grammar. Each transition is equipped by a set containing all the required features in its concurrent transitions (Fig.23).

```
in Feature_FunctMod.txt
fmod FTS_FunctMod is
protecting Feature_FunctMod .

sorts TsState FtsState .
ops State1 State2 State3 State4
State5 State6 State7 State8 : -> TsState [ctor].
op < _ ; _ ; _ > : TsState FeatSet Bool -> FtsState .

endfm
```

Figure 22. The functional module *FTS_FunctMod*

For example, Fig.22 shows that the attribute *Set_ReqFeatCTs* of the transition soda contains the features c and t which are exactly the features required in the concurrent transition cancel and tea.

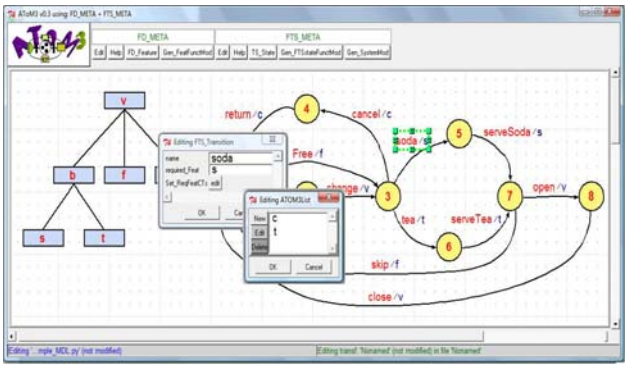


Figure 23. Decorated FTS model

Finally, to generate automatically the system module *FTS_SysMod* (Fig.24), we have to execute the *Gen_SystemMod* graph grammar. It uses, as input, the enriched FTS and FD models obtained by applying the first two graph grammars.

```
in FTS_FunctMod.txt
mod FTS_SysMod is
including FTS_FunctMod .

var ListSelectFeats : FeatSet .
crl [pay] : < State1; ListSelectFeats ;false > => < State2 ; ListSelectFeats;false >
if Isin( v , ListSelectFeats) and ( not Isin( f , ListSelectFeats )
or ( Isin( f , ListSelectFeats ) and not Isin( f , c f s t b ) ) ) .
crl [change]:< State2; ListSelectFeats ;false > => < State3; ListSelectFeats;false >
if Isin( v , ListSelectFeats ) .
crl [free] : < State1; ListSelectFeats ; false > => < State3; ListSelectFeats ; false>
if Isin( f , ListSelectFeats ) and ( not Isin( v , ListSelectFeats )
or ( Isin( v , ListSelectFeats ) and not Isin( v , empty ) ) ) .
crl [return]:< State4; ListSelectFeats ; false > => <State1; ListSelectFeats ; true >
if Isin( c , ListSelectFeats ) .
crl [cancel]:< State3; ListSelectFeats; false > => <State4 ; ListSelectFeats;false >
if Isin( c , ListSelectFeats) and ( not Isin( s , ListSelectFeats )
or ( Isin( s , ListSelectFeats ) and not Isin(s , empty) ) )
and ( not Isin( t , ListSelectFeats)
or ( Isin( t , ListSelectFeats) and not Isin( t , empty) ) ) .
crl [soda]:< State3; ListSelectFeats ; false > => <State5 ; ListSelectFeats;false >
if Isin( s , ListSelectFeats ) and ( not Isin( t , ListSelectFeats )
or ( Isin( t , ListSelectFeats ) and not Isin(t , empty ) ) )
and ( not Isin( c , ListSelectFeats)
or ( Isin( c , ListSelectFeats) and not Isin( c , empty) ) ).
crl [tea] : < State3; ListSelectFeats ; false > => < State6; ListSelectFeats ;false >
if Isin( t , ListSelectFeats ) and ( not Isin( c , ListSelectFeats )
or ( Isin( c , ListSelectFeats ) and not Isin(c , empty) ) )
and ( not Isin( s , ListSelectFeats)
or ( Isin( s , ListSelectFeats) and not Isin( s , empty) ) ) .
crl [serveSoda]:<State5;ListSelectFeats ;false> => <State7;ListSelectFeats;false>
if Isin( s , ListSelectFeats ) .
crl [serveTea]:<State6; ListSelectFeats ;false > => < State7;ListSelectFeats;false>
if Isin( t , ListSelectFeats ) .
crl[open]:< State7; ListSelectFeats ; false > => < State8 ; ListSelectFeats;false >
if Isin( v , ListSelectFeats ) and ( not Isin( f , ListSelectFeats )
or ( Isin( f , ListSelectFeats ) and not Isin( f , c f s t b ) ) ) .
crl [skip]:< State7; ListSelectFeats ; false > => < State1 ;ListSelectFeats ; true>
if Isin( f , ListSelectFeats ) and ( not Isin( v , ListSelectFeats )
or ( Isin( v , ListSelectFeats ) and not Isin( v , empty ) ) ).
crl [close]: < state8; ListSelectFeats; false> => < state1 ; ListSelectFeats; true >
if Isin( v , ListSelectFeats ) .

endm
```

Figure 24. The system module *FTS_SysMod*

After obtaining the generated SPL Maude specification by applying these graph grammars, we can now move to simulation and analysis.

A. *Simulation*

As an initial state of the FTS, we use:

< State1 ; v b s f ; false >

In Fig.25, we show the simulation of the resulting Maude specification.

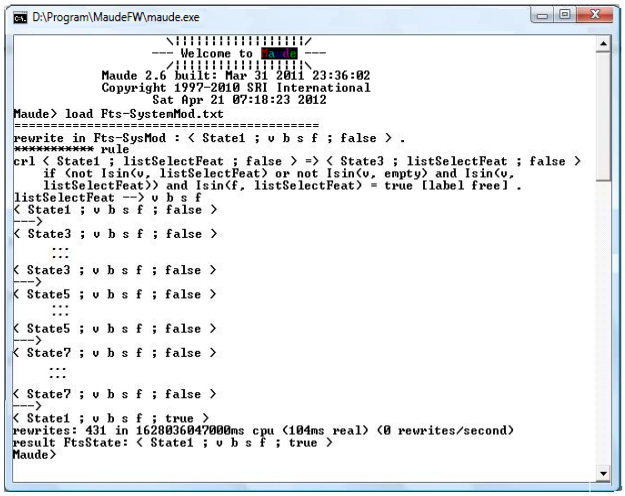


Figure 25. Results of simulation

According to the simulation results, we see that the priority relation between transitions is preserved.

B. *Verification and Analysis*

For the verification and analysis purpose, we have to define manually properties to verify. This section illustrates the use of Maude's LTL model checker. Consider the vending machine SPL example. A temporal property, that each valid product must satisfy, is the fact that, from the initial state, the system always finishes in a final state. Properties are expressed using predicates. First, using the Boolean flag of the FTS state, we define two predicates *Initial* and *Final* in a new module called *FTS_PredicatesMod* as follows:

```
in model-checker
in FTS_SysMod.txt
mod FTS_PredicatesMod is

protecting FTS_SysMod   .
including SATISFACTION .

subsort FtsState <  State .
op Initial    : TsState -> Prop .
op Final      : TsState -> Prop .

var State  : TsState .
var ListSelectFeats    : FeatSet .
var flag : Bool .

ceq  < State ; ListSelectFeats ; flag >  |= Initial (State) = true
                                        if ( flag == false and State == State1) .
ceq  < State ; ListSelectFeats ; flag >  |= Final  (State) = true
                                        if ( flag == true  and State == State1) .
endm
```

Figure 26. *FTS-PredicatesMod* module

The latter property is expressed in LTL as:

[ ] (Initial (State1) |-> Final (State1))

Now, consider two variants of the vending machine SPL:

P_1 { v , b , s } and P_2 { v , b , f }.

The following module verifies the propriety on P_1 and P_2:

```
in FTS_PredicatesMod.txt
mod FTS_Check is
protecting FTS_PredicatesMod .
including MODEL-CHECKER .
including LTL-SIMPLIFIER .

ops FTS_Init1   FTS_Init2  : ->  FtsState  .
eq  FTS_Init1   =  < State1 ;  v  b  s  ;  false > .
eq  FTS_Init2   =  < State1 ;  v  b  f  ;  false > .
endm

red modelCheck ( FTS_Init1 , [ ] ( Initial (State1) |-> Final (State1)) ) .
red modelCheck ( FTS_Init2 , [ ] ( Initial (State1) |-> Final (State1)) ) .
```

Figure 27. LTL property to check

Maude's LTL model checker results are:

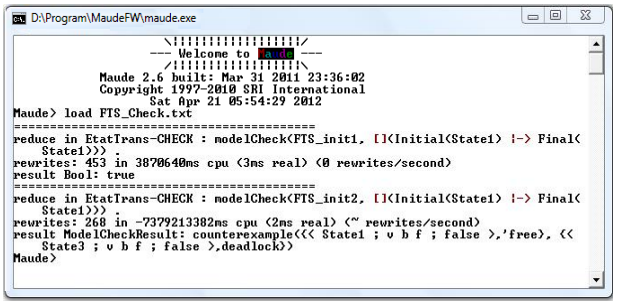


Figure 28. Property verification results

The results show that the property is successfully verified for the product P_1 {v, b, s}. For the product P_2 {v, b, f}, the property does not hold and a counterexample path is displayed.

Using the specification proposed in this work, other temporal properties can be verified.

C. *Discussion*

The proposed approach has many advantages. The most important are:

- Our framework is fully automated.
- Our approach considers that the SPL can evolve. In case of updating the source models, the correction of the Maude specification will be automatically made.
- According to Classen et al. [1], if the modelled SPL consists of several processes running in parallel, each process can be modelled as a separate FTS, all sharing the same FD. The FTS of the system is obtained by composing these processes. As Maude offers great possibilities for parallel programming, our approach allows composition.

VIII. CONCLUSION

Research in the field of SPL is becoming increasingly important, particularly through its ability to increase software reuse. Over the past few years, several modelling and analysis techniques have been published.

In this paper we proposed and implemented a graph transformations and rewriting logic based framework for SPLs specification and analysis. The basic idea is to automatically translate FTS and FD models into their equivalent Maude specification by applying three

proposed graph grammars. Transitions that express the dynamic of an FTS are directly translated into rewriting rules. The priority between alternative transitions is expressed in the conditions of rewriting rules by using FD model. The power of this specification resides in the fact that the transformation preserves FTS and FD semantics. The result procures a formal description that offers a solid basis for the verification process. The rewriting logic language Maude is used. Its LTL model checker has allowed verifying temporal properties. Thus, verification of the individual behavior of each product is guaranteed and therefore we can identify the products that violate the required properties.

As Maude offers great possibilities for parallel programming, our approach allows composition of FTSs. We consider this work as a new way of investigation in SPLE domain. It combines the advantages of both, graph transformations and rewriting logic into an automatic framework.

In Software Product Line engineering, a Feature Diagram defines features and their relationships. Each product is defined as a combination of features. For a given valid product, dependencies that have each feature with the others must be respected. As example, mandatory feature must be selected whenever its parent is selected. Note that in this paper, the proposed method allows to check the behavior of a given product without worrying about these dependencies. In a future work, we plan to extend our framework to be able to check automatically the validity of products according to this perspective.

ACKNOWLEDGMENT

We would like to thank the referees for their helpful comments and suggestions.

REFERENCES

[1] 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″, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol.1, pp. 335–344, May 2010.

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

[3] J. Meseguer, ″Conditional rewriting logic as a unified model of concurrency″, Theoretical Computer Science, vol. 96(1), pp. 73-155, 1992.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada, ″Maude : specification and programming in rewriting logic″, Internal Report, SRI International, 1999.

[5] K. Khalfaoui, A. Chaoui, C. Foudil and E. Kerkouche, ″Specification of software product lines in rewriting logic″, Internal Report MFGL-01-11, Misc Laboratory, Department of Computer Science, University of Constantine, Algeria, 2011, unpublished.

[6] M. Andries, G. Engels, A. Habel, B. Hoffmann, H. J. Kreowski, S. Kuske, D. Pump, A. Schürr and G. Taentzer, ″Graph transformation for specification and programming″, Science of Computer Programming, vol. 34(1), pp. 1-54, April 1999.

[7] J. De Lara and H. Vangheluwe, ″AToM3: a tool for multi-formalism modelling and meta-modelling″, Lecture Notes in Computer Science, vol. 2306, pp.174-188, April 2002.

[8] P. Clements and L. Northrop, ″Software product lines: practice and patterns″, Addison-Wesley, 2001.

[9] K.G. Larsen, U. Nyman and A. Wasowski, ″Modal I/O automata for interface and product line theories″, Lecture Notes in Computer Science, vol. 4421, pp. 64–79, 2007.

[10] D. Fischbein, S. Uchitel and V. Braberman, ″A foundation for behavioural conformance in software product line architectures″, Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA 06), ACM Press, pp. 39–48, 2006.

[11] A. Fantechi and S. Gnesi, ″Formal modeling for product families engineering″, Proceedings of the 12th International Software Product Line Conference (SPLC'08), IEEE Computer Society Press, pp. 193–202, 2008.

[12] P. Asirelli, M. H. ter Beek, S. Gnesi and A. Fantechi, ″A deontic logical framework for modelling product families", Fourth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'10), ICB Research Report, vol.37, pp. 37–44, 2010.

[13] H.C. Li, S. Krishnamurthi and K. Fisler, ″Verifying cross-cutting features as open systems″, Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering, vol. 27(6), pp. 89–98, 2002.

[14] K. Lauenroth, S. Toehning and K. Pohl, ″Model checking of domain artifacts in product line engineering″, Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, pp. 269–280, 2009.

[15] J. De Lara and H. Vangheluwe, ″Computer aided multi-paradigm modelling to process Petri-nets and Statecharts″, Lecture Notes in Computer Science, vol. 2505, pp. 239-253, October 2002.

[16] E. Kerkouche, A. Chaoui, E. B. Bourennane and O. Labbani, ″On the use of graph transformation in the modeling and verification of dynamic behavior in UML models″, Journal of Software, vol. 5(11), pp. 1279-1291, 2010.

[17] R. Elmansouri, H. Hamrouche and A. Chaoui, ″From UML Activity diagrams to CSP expressions: a graph transformation approach using AToM3 tool″, International Journal of Computer Science Issues, vol. 8(2), pp. 368-374, 2011.

[18] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C.Talcott, ″Maude manual (version 2.2)″, Internal Report, SRI International, December 2007.

[19] A. Pnueli, ″The temporal logic of programs″, Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS), pp. 46–57, 1977.

[20] S. Eker, J. Meseguer and A. Sridharanarayanan, ″The Maude LTL model checker″, Proceedings of the 4th International Workshop on Rewriting Logic and Its Applications (WRLA), Electronic Notes in Theoretical Computer Science, vol. 71, 2002.

[21] G. Rozenberg, ″Handbook of graph grammars and computing by graph transformation″, World Scientific, Singapore, vol. 1, 1999.

[22] J. De Lara, J. Vangheluwe, and M. Alfonseca, ″Meta-modelling and graph grammars for multi-paradigm modelling in AToM3″, Software and Systems Modelling, Springer Verlag, vol 3(3), pp. 194-209, August 2004.

[23] Python Home page, http://www.python.org

Khaled Khalfaoui is Assistant Professor in the department of computer science, University of Jijel, Algeria. His research field is Software Product Lines, graph transformations and formal methods.

Allaoua Chaoui is Professor in the department computer science, University Mentouri Constantine, Algeria. His research interests include mobile computing, formal specification, verification of distributed systems and model transformations.
Foudil Cherif is Associate Professor in the department of computer science, University of Biskra, Algeria. His research field is complex systems, artificial life and behavioral simulation.
Elhillali Kerkouche is Associate Professor in the department of computer science, University of Jijel, Algeria. His research field is formal methods and distributed systems.