On the use of Algebraic Petri Nets for Software Product Lines Specification: a Graph Transformation Based Approach

Khaled Khalfaoui
Department of Computer Science, University of Jijel, Jijel, Algeria

Allaoua Chaoui
MISC Laboratory, Department of Computer Science, University of Constantine, Algeria

Cherif Foudil
Department of Computer Science, University of Biskra, Biskra, Algeria

Elhillali Kerkouche
Department of Computer Science, University of Jijel, Jijel, Algeria

Abstract: A Software Product Line is a set of software products that share a number of core properties but also differ in others. The variants of a Software Product Line are defined and implemented in terms of features, which are subsequently combined in specific ways to obtain the final software products. In this context, formal modelling is critical for managing the inherent complexity of systems with a high degree of variability. Algebraic Petri Nets are a powerful formalism to model complex systems. It combines the expression power of Petri nets and abstract data types. In this paper, we propose an automatic approach to map Software Product Line models whose products behaviours are modeled with a featured transition system into an equivalent Algebraic Petri Nets. The translation process is based on graph transformations. An illustrative example is presented. By this, we obtain a formal specification allowing several analysis and verification activities.

Keywords: Software Product Line, Featured Transition System, Formal Modelling, Algebraic Petri Nets, Graph Transformations.

1. Introduction

A Software Product Line (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 [1]. Research in the field of SPLs is becoming increasingly important, particularly through its ability to increase software reuse. However, due to the inherent complexity of systems designed following this paradigm, it is hard to verify their correctness without a rigorous formal modeling.

Algebraic Petri Nets (APNs) are a kind of net/data model combining the strengths of Petri Nets with those of abstract data types [2]. Petri nets are used for their foundation in concurrency and dynamics, while abstract data types are used for their data abstraction power and solid theoretical foundation. The power of this formalism lies mainly in its richness in term of expression. In addition to modelling, this formalism allows a formal verification and analysis.

In software engineering, graph transformation techniques are widely used. Generally, the aim is to transform system graphical models into their formal equivalent specifications supporting assessment and analysis of characteristics. This task is performed by executing a graph grammar [3]. To manipulate syntactically correct models according to the formalisms definition, meta-modeling technique is necessary.

In this work, we develop an automatic framework based on graph transformations to map the Featured Transition System (FTS) diagrams [4] into an equivalent APNs model. This paper is organized as follows. Section 2 outlines some related works. In section 3, we recall some basic notions about the FTS formalism. We give an overview of APNs in section 4. Section 5 summarizes briefly the pertinent concepts of graph transformation and the AToM³ tool [5]. Section 6 presents our approach. We first introduce the idea behind the translation. Then, we define the used meta-models and two graph grammars. In section 7, we illustrate the framework through an example. Finally, section 8 concludes the paper and gives some perspectives of this work.

2. Related Works

Software Product Line Engineering (SPLE) is an approach for developing families of software systems. In this area of research, several modelling and analysis techniques have been published. Larsen et al. in [6] propose modal I/O automata to model variability in component interfaces and discuss compatibility.
between these interfaces. In a similar effort, Fischbein et al. [7] propose Modal Transition Systems (MTS) to model SPLs and examine the notions of behavioural conformance in MTS that are suitable for SPLs. Fantechi et al. [8] extended their approach by introducing explicit variability operators into MTS. In [9] Asirelli et al. apply deontic logic to express both static and behavioral aspects of product families. Classen et al. in [4] had introduced the FTS formalism that allows modeling the combined behaviour of a whole system family with a single parameterized model.

Nowadays, graph grammars are widely used for modelling and analysis of complex systems in the area of software engineering. De Lara et al. [10] propose a translation of Statecharts into Petri Nets based on graph grammars. The authors in [11] have proposed a tool that formally transforms the dynamic behaviour of systems expressed using UML Statechart and collaboration diagrams into their equivalent Colored Petri Nets (CPN) models. For analyzing purposes, Boudiaf in [12] had proposed an automatic translation of Ada programs towards ECATNets formalism.

The current paper presents an automatic approach based on graph transformations for the specification of SPL models using APNs formalism. The obtained result offers a solid basis for the simulation and verification activities.

3. Variability Modeling

For SPLs modeling, several techniques have been proposed. In this work, we are interested in FTS formalism [4]. The combined behaviour of a whole system family is described using a transition system (TS) associated to a Feature diagram (FD) [13].

The FD model is used to express the structural view of the SPL. It is a graphical representation which shows a hierarchically structured set of features of the product line.

![Feature Diagram](image)

**Figure 1 — Feature Diagram**

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). As an example, Fig.1 shows the FD of a vending machine SPL inspired from [4].

FTS is a TS in which each transition is marked with the required feature in addition to being labelled with actions, and where a priority relation is associated to transitions leaving the same state. The FTS of the vending machine is presented in Fig.2.

**Figure 2 — Featured Transition System**

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 all transitions linked to features that are not in this product.
- Removing all transitions that are overridden by higher priority transitions.

The result of the projection is an ordinary TS.

4. Algebraic Petri Nets

Algebraic Petri Nets [14] is a formalism that combines Petri nets and abstract algebraic data types (AADT). An APNs is a tuple $\langle \text{Spec}; T; P; AX \rangle$ where:

- Spec = $\langle S,F,X,E \rangle$ is an algebraic specification with:
  - $S$ is a set of sorts.
  - $F$ is a set of operations symbols.
  - $X$ is a set of variables indexed by $S$.
  - $E$ is a set of equations.
- $T$ is a set of transitions.
- $P$ is a set of places indexed by $S$.
- $AX$ is a set of axioms. It is composed of axioms of form $\langle t, \text{cond in, out} >$ with:
  - $t$ is the transition.
  - $\text{cond}$ is the condition for firing of transition. This term is a boolean.
  - $\text{in and out}$ represent pre and post-conditions of the transition.

This formalism has two aspects:

- The control part which is handled by a Petri Nets.
- The data part which is handled by AADTs.

For a given transition, the firing conditions are called guard. Consider the example in the following figure:
The firing of this transition takes a random token from \( P_1 \) for which the value is less than 5 (guard), adds two units to its value and places it in \( P_2 \). An assignment of the variable \( N \) is realized (for more details see [2]).

The main advantage of APNs resides in the fact that complex systems such as SPLs can be described in a condensed way. In addition to this power of expression, APNs provide a solid basis for the verification process. In the last years, several automated analysis tools have been developed such as ALPiNA [15].

5. Graph transformation

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. Graph transformation rules are usually called graph grammars [3]. 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 sub-graph 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. 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.

In this field of research, meta-modeling technique is widely used to manipulate syntactically correct models according to formalisms needed in the 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. On the other hand, the concrete graphical syntax to define graphical appearance of these entities and relationships.

AToM$^3$ [5] is a graph transformations tool implemented in Python. 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. The use of global attributes available in all of the graph grammar rules as well as constraints is allowed. In AToM$^3$, the meta-formalism used is the Entity-Relationship diagram.

6. Proposed approach

In this section, we present our technique used for the specification of the SPL models using the APNs formalism. At the beginning, it is preferable to begin by introducing the idea behind the translation.

6.1. Formalization

In APNs, all manipulated elements are specified using AADTs. For this, we define:

- A sort called Feature.
- A sort called ListFeatures to manipulate sets of features.
- A boolean function called Contains \((\text{listFeats}, f_i)\) to check the membership of a feature to a set of features.

Consider the transition \( \text{Trans}_i \) (Fig.5):

To generate its equivalent APNs transition, there are two possible configurations:

**Configuration:** when there is no need to verify the priority of the required feature \( f_i \) over all features required in the concurrent transitions. This is the case where:

- \( \text{Trans}_i \) is the only one output transition from the source state (State).
- Or else, for each feature \( f_k \):
  - \( f_i \) is a higher priority than \( f_k : f_i \) is a descendant of \( f_k \) in FD model.
  - Or \( f_i \) and \( f_k \) are independents. There is no relationship between the two features in FD model.
In the FTS formalism, this transition is enabled, when the feature f_i is in the set of selected features (SelectFeats) of the SPL considered product. We propose the equivalent APNs transition presented in Fig.6.a.

**Configuration:** when we have to check the priority of the required feature f_i over features required in the concurrent transitions. This is the case when there exists at least one feature f_k descendant of f_i in the FD model. Here, according to the semantics of FTS formalism, the transition Trans is enabled when two conditions are simultaneously satisfied:
- The feature f_i is in SelectFeats.
- All features f_k required in the concurrent transitions which are simultaneously descendants of f_i in the FD model are not in the set SelectFeats.

For this configuration, we propose the equivalent APNs specification presented in Fig.6.b.

```
(a)
SelectFeats   SelectFeats
P              P'
Contains (SelectFeats, f_i)

(b) Trans_i
SelectFeats   SelectFeats
P              P'
Contains (SelectFeats, f_i) and not (Contains (SelectFeats, f_k))
```

Figure 6 — Equivalent APNs transition

In both configurations, the inscription of InputArcs and OutputArcs is SelectFeats. To generate guards of transitions, the code verifying the presence of the required feature (Contains (SelectFeats, f_i)) is always generated in the same way. To generate the code specifying the priority conditions, we have to calculate at first the set containing all required features in the concurrent transitions that are descendents of the required feature f_i. Then, in case where this set is not empty (configuration1), we run through all elements of this set and for each one (f_k) we have to add the code:

```
and not (Contains (SelectFeats, f_k))
```

Else, one has nothing to add (configuration2).

In resume, to translate the FTS models into an equivalent APNs, we have to create for each transition:
- Two APNs places: representing its source and destination states in the FTS model.
- An APNs transition: the guard will be generated as proposed previously.
- An InputArc and an OutputArc: for both the inscription is SelectFeats.

### 6.2. Automatic Translation

Now return to the automatic generation of the APNs model using the graph transformation approach. We propose a process with two steps (Fig.7). Briefly, the first step consists of meta-modeling FD, TS and APNs formalisms to generate automatically a visual modeling tool for each of them using AToM3. The second step is to define the used graph transformations.

![Image](https://via.placeholder.com/150)

Figure 7 — The general outline of the proposed approach

In the following subsections, we will give details of each step.

**Meta-modeling**

To implement the proposed solution, we add some additional attributes in our FD and TS meta-models.

**FD meta-model:**

FD models consist of nodes and links between these nodes (Fig.1). We propose a meta-model called FD-MetaModel as follows:

- **FD-Feature Entity:** This class is used to represent features. It has two attributes: its name and the set of all its descendents called Set_DescFeats.

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

**TS meta-model:**

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

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

- **TS-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). The latter is proposed to contain all the features required in the transitions leaving the same state as this transition.

**APNs meta-model:**
Since APNs consist of places, transitions, arcs from places to transitions and arcs from transitions to places, they are meta-modeled using two entities and two relationships described as follows:
- **APNs-Place Entity:** each place has two attributes: its identifier (name) and the marking.
- **APNs-Transition Entity:** It has two attributes: its name and the firing conditions (guard).
- **APNs-InputArc Relationship:** It has only one attribute: inscription.
- **APNs-OutputArc Relationship:** It has only one attribute: inscription.

To fully define our meta-models, we have also specified the graphical appearance of each entity of the FD, TS and APNs formalisms according to its appropriate notation.

**Defining the Graph Grammars**
To make the transformation easier, we propose to use two complementary graph grammars. The first graph grammar computes the added attributes in FTS models. Then, the second graph grammar generates the equivalent APNs model.

1st GG: **Decorating FTS models**
The first purpose of this grammar is to produce the set containing all descendents (Set_DescFeats) for each feature in the FD diagram. Treatment begins with the leaves. Each time the rewriting system locates a leaf and adds his feature to the attribute Set_DescFeats of its parent. Then, we move on to intermediate nodes. At this level, the treatment only applies to nodes for which all children were treated. Similarly, for each one, the graph grammar adds its feature and its descendents to the attribute Set_DescFeats of its parent and so forth. At last, we treat the root node. To do this, we use two attributes: Visited to indicate whether the node has been previously treated or not and Count_PrChild to count the number of its treated children. To perform this treatment, we propose in this graph grammar three rules (Fig.8). 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 to 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 descendents will be added to its parent Set_DescFeats attribute. To avoid this process 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.

![Figure 8 — Decorating FD model Rules](image)

The second purpose of this graph grammar is to work out for each feature of the TS model the attribute Set_ReqFeatCTs. The idea behind this decoration is to pass through the states one by one. For the treated state, 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 auxiliary attributes for states, Current and Visited. The Current attribute is used to identify the state in the TS 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. To carry out this process, we
propose seven rules in our graph grammar (Fig.9). These rules are described as follows:

**Add-CF2SetReqFeatCTs (priority 4):** 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 5):** 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 6):** this rule is applied to initialize the FeatureInserted attribute of all output transitions of the current state to process another transition which is not yet treated.

**SelectTransition: (priority 7):** 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. Subsequently, rules N°4, N°5 and N°6 will be triggered.

**ProcessSingleOutputTransition (priority 8):** This rule deals with the case where the current transition is the single output transition from the current state. It marks this transition as visited and its attribute Set_ReqFeatCTs is set empty. In this case, rules N°4, N°5, N°6 and N°7 are not applied.

**Set-ProcessedStateAsVisited (priority 9):** 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 10):** is applied to select a state from TS model that has not been previously visited to produce the Set_ReqFeatCTs attribute of all its output transitions.

---

**Figure 9 — Decorating TS model Rules**

---

2nd GG: FTstoAPNs

Once the attributes Set_DescFeats and Set_ReqFeatCTs calculated, one passes to the automatic generation of the APNs model. To do this, we propose to traverse the TS model through its transitions from the initial states to next states and so forth. For this reason, in TS states we add a temporary attribute called Visited to indicate whether each initial TS state has been yet treated or not.
Our graph grammar is composed of five rules (Fig.10).

1- **Treat-InitialStates**:
   
   **LHS**
   
   ![Initial States Graph](image1)

   **RHS**
   
   ![Specified Places Graph](image2)

   **CONDITION**
   
   node(2).visited == False and node(2).initial == True

   **ACTION**
   
   node(1).visited = 1

2- **TreatTrans-SourDestPlacesCreated**:

   **LHS**
   
   ![Sour Dest Places Graph](image3)

   **RHS**
   
   ![Specified Places Graph](image4)

   **CONDITION**
   
   node(4).required_Feat == node(3).name

   **ACTION**
   
   node(5).name = node(3).name

3- **TreatTrans-DestPlaceNotCreated**:

   **LHS**
   
   ![Dest Place Not Created Graph](image5)

   **RHS**
   
   ![Specified Places Graph](image6)

   **CONDITION**
   
   node(3).required_Feat == node(5).name

   **ACTION**
   
   node(7).name = node(3).name

4- **Delete-TSstate**:

   **LHS**
   
   ![TS State Graph](image7)

   **RHS**
   
   ![Empty Graph](image8)

   **CONDITION**
   
   node(2).required_Feat == node(3).name

   **ACTION**
   
   node(1).name = node(3).name

5- **Delete-Feature**:

   **LHS**
   
   ![Feature Graph](image9)

   **RHS**
   
   ![Empty Graph](image10)

   **CONDITION**
   
   node(3).required_Feat == node(1).name

   **ACTION**
   
   node(2).name = node(3).name

7. **Illustrative Example**

To illustrate our framework, let us consider the vending machine example which was seen previously (Section 3). As input, we have to create the FTS models as shown in Fig.11.

By executing the first graph grammar, we obtain:

- A decorated FD model: each feature is enriched with the set of its descendants.
- A decorated TS model: each transition is equipped by a set containing all the required features in its concurrent transitions.

The results are presented in Fig.12. It 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.
- The attribute `Set_ReqFeatCTs` of the transition soda contains the features t and c which are exactly the features required in the concurrent transitions tea and cancel.

To generate the equivalent APNs model, we have to execute the second graph grammar on the enriched FD and TS models. Fig.13 shows the obtained result.

---

**Figure 10 — FTSToAPNs Graph Grammar**

**Treat-InitialStates(priority1):** this rule is applied to treat the initial states in the TS model. Each time, it locates a TS initial state to associate it an APNs place. This rule serves to initialize the translation process. **TreatTrans-SourDestPlacesCreated(priority2):** this rule is applied to treat transitions for which the two APNs places associated to the source and the destination state are already created. **TreatTrans-DestPlaceNotCreated(priority 3):** this rule is applied to treat transitions for which only the APNs place associated to the source state is already created. **Delete-TSstate(priority4):** this rule is applied to clean the obtained APNs of the TS model states. Each time, it locates a state and deletes it. **Delete-Feature(priority5):** this rule is applied to clean the obtained APNs of the FD model features. Each time, it locates a Feature and deletes it.
Let us take as example, the transition “pay”. Its guard is: Contains (SelectFeats, v ) and 
(not Contains (SelectFeats, f )
This transition is enabled when the set of the selected features contains its required feature v and does not contain the feature f. This is the fact that f is higher priority than v according to the feature diagram FD.

8. Conclusion

Software product line engineering is an approach for developing families of software systems. The main advantage over traditional approaches is that all products can be developed and maintained together.

In this paper, we have presented an automatic framework to translate FTS models into an equivalent APNs. We have proposed two complementary graph grammars. Priority between alternative transitions is expressed in the guards of the generated APNs transitions. The resulting specification expresses the behaviour of the entire SPL family and individual product behaviour can be derived by giving the set of its specific features. According to the obtained results, FTS semantics are preserved.

The choice of APNs formalism is motivated by its formal semantics. It offers a solid basis for the simulation and verification activities. Once the equivalent APNs generated, model checking technique allows us verifying correctness properties of the SPL variants.

In the future, we plan to extend this framework to be able to analyze automatically the generated specification. This requires another transformation to obtain an APNs description interpretable by an automatic algebraic Petri nets analyzer.

References