Extended UML stratecharts and product lines

González, Ariel
Luna, Carlos

Documento de Investigación No. 7
Facultad de Ingeniería
Universidad ORT Uruguay
Setiembre, 2009
ISSN 1688-8707

Documento de Investigación

UNIVERSIDAD ORT
Uruguay

ISSN 1688-8707

Extended UML stratecharts and product lines

Ariel González (UNRC, Argentina)
Carlos Luna (Facultad de Ingeniería Universidad ORT Uruguay)

Documento de Investigación No. 7
Facultad de Ingeniería
Universidad ORT Uruguay
Setiembre, 2009

Extended UML Statecharts and Product Lines

Ariel Gonzalez
UNRC, Argentina
`gonzalezg@exa.unrc.edu.ar`

Carlos Luna
Universidad ORT, Uruguay,
`luna@ort.edu.uy`

Abstract

The study of variability in software development has become increasingly important in recent years. The research areas in which this is involved range from software specialization to product lines. A common mechanism to represent the variability in a product line is by means of feature models. However, the relationship between these models and UML design models is not straightforward. UML statecharts are extended introducing variability in their main components, so that the behavior of product lines can be specified. The contribution of this work is the proposal of a rule-based approach that defines a transformation strategy from extended statecharts to concrete UML statecharts. This is accomplished via the use of feature models, in order to describe the common and variant components, in such a way that, starting from different feature configurations and applying the rule-based method, concrete state machines corresponding to different products of a line can be obtained.

1. Introduction

A product line (PL), also called system family, is a set of software 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 [1,2,3].

To develop a system family as opposed to developing a set of isolated systems has important advantages. One of these is the possibility of building a kernel that includes common features, from which desired products can be easily built as extensions. By adding distinguishing characteristics (variability) to such a kernel, different products can be obtained [3,4,5,6,7,8]. For example, today we observe in the market a significant number of different types of mobile phones (MPs) that share a core of basic features and differ in other more specific characteristics: availability of a digital camera, internet access, mp3 player, among others.

The UML language [9] provides a graphical notation and has become the standard for modeling different aspects of software systems. Statecharts and interaction diagrams are part of the set of tools that UML provide so that the system behavior can be specified, which are specially suitable for the software design phase. Statecharts are used to specify the behavior of the instances of a class (intra-component behaviour), and therefore constitute an appropriate mechanism for describing the behavior of certain problems by means of a graphical representation. In the latest version of UML 2.0, the statecharts do not offer operators and/or sublanguages for specifying system family.

In this paper we propose an extension of UML statecharts for modeling PLs. We used feature models so that both common and variant functionality of a system family can be described [10,8], and we incorporate variability in the essential components of the statecharts, in such a way that, starting from different configurations of a feature model, concrete statecharts corresponding to different products of a PL can be generated applying a rule-based method. The approach defines the transformation strategy from extended statecharts to concrete UML statecharts.

The rest of the work is organized as follows. In section 2 and 3 we briefly introduce statecharts and feature models, respectively. Section 4 presents an extension of statecharts with variant elements, which together with the use of feature models allow specifying PLs. In section 5 we detail the mechanisms for obtaining products of a PL from distinct configurations of a feature model, via the use of rules and a transformation strategy. Related works are discussed in section 6 and finally, we conclude and discuss possible further works in section 7. We exemplify the proposed work by developing part of a case study based on mobile phone technology. A preliminary version of this work is [11]. As opposed to [11], this paper presents a rule-based approach. We define an application strategy for the rules in a proper manner, in such a way that inconsistencies are avoided in the statechart obtained. The rules are organized in a sequence of rule sets, in which each rule set can be considered as a layer. Within a rule set, the rules may be applied in a non-deterministic order [12]. We also formalized and added rules that in [11] are omitted or only described informally.

2. Statecharts

UML StateCharts (SCs) constitute a well-known specification language for modeling the dynamic system behavior. SCs were introduced by D. Harel [13] and later incorporated in different versions of the UML with some variations. In this section, we present definitions of SCs based on [14]. For additional details, the reader is referred to [11,14].

SCs consist essentially of states and transitions between states. The main feature of SCs is that states can be refined, defining in this way a state hierarchy. A state decomposition can be sequential or parallel. In the first case, a state is decomposed into an automata (Or-state). The second case is supported by a complex statechart composed of several active sub-statecharts (And-state), running simultaneously. In figure 1, Or-state s0 is the highest state in the hierarchy, which is composed of four states, namely: s1, s2, s3 and s8. On the other hand, state s3 is composed of the parallel combination of s4 and s7. A SC configuration represents the state of the system at a given instant in time, which is characterized by the set of active states.

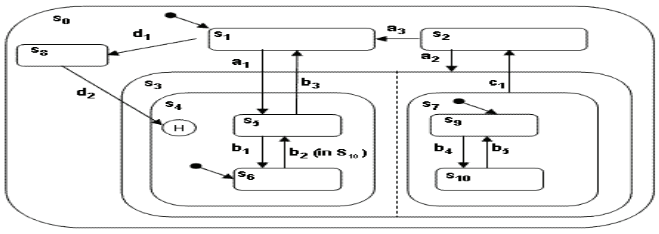

Figure 1. Graphical representation of a SC.

Let *S, TR, Π* and *A* $(\Pi \subseteq A)$ be countable sets of state names, transition names, events and actions of a SC, respectively. Also, let us define $s \in S$ as either a basic term of the form $s = [n]$ (Simple-state), as a term Or of the form $s = [n, (s_1 ,..., s_k), l, T]$ (Or-state), or as a term And of the form $s = [n, (s_1 ,..., s_k)]$ (And-state), where $name(s) = n$ is the name of the state *s*. Here $s_1 ,..., s_k$ are the subterms (substates) of *s*, also denoted by $sub\_state(s) = (s_1 ,..., s_k)$. Likewise, $initial(s) = s_1$ is the initial state of *s*, $T \subseteq TR$ is the set of internal transitions of *s*, and *l* the active state index of *s*. A transition is represented as a tuple $t = (t', s_o, e, c, \alpha, s_d, ht)$, where $name(t) = t'$ is the transition name, $source(t) = s_o$ and $target(t) = s_d$ are called source and target of t, respectively, $ev(t) = e$ the trigger event, $cond(t) = c$ the trigger condition, and $acc(t) = \beta$ is the sequence of actions that are carried out when a transition is triggered. In addition, $hist(t) = ht$ is the history type of the target state of *t* [14]. The graphical notation used in the transitions is $t : e,c/\beta$.

3. Feature Models

Feature Models (FMs) are used to describe properties or functionalities of a domain. A functionality is a distinctive characteristic of a product or object, and depending of the context it may refer to, it is a requirement or component inside an architecture, and even code pieces, among others. FMs allow us to describe both commonalities and differences of all products of a PL and to establish relationships between common and variant features of the line. There are multiple notations for describing FMs. In this work we will use the proposal of Czarnecki [10].

A tree structure instance is a *FM configuration* (*FMConf*) that describes the model and that respects the semantics of their relations. That is, a FM allows one to identify common and variant features between products of a PL, while a FM configuration characterizes the functionalities of a specific product. Formally, the concepts of FMs are defined as follows:

Definition 1. A *FM* is defined as a tree structure represented by a tuple *(Funcs,* f_0*, Mand, Opt, Alt, Or-rel)*, where *Funcs* is a set of functionalities of a domain (nodes of the tree), $f_0 \in Funcs$ is the root functionality of the tree and, *Mand*, *Opt*, *Alt*, *Or-rel* $\subseteq$ *Funcs* x $(\wp(Funcs)\text{-}\{\varnothing\})$ the mandatory, optional, alternative and disjunct relations of the model, respectively. If $(f, sf) \in Mand \cup Opt$, $\#sf = 1$.

Definition 2. A *FM configuration* corresponding to a *FM (Funcs, f_0, Mand, Opt, Alt, Or-rel)* is a tree *(F, R)* where *F* is the set of nodes and *R* the set of edges; $F \subseteq Funcs$ and $R \subseteq \{(f,sf) \in F \times (\wp(F) - \{\varnothing\}) \;/\; \exists sf' \in \wp(Funcs)\colon sf \subseteq sf' \wedge (f, sf') \in Mand \cup Opt \cup Alt \cup Or\text{-}rel\}$. Moreover, the following conditions must be fulfilled by *(F, R)*: (1) $f_0 \in F$; (2) for every $(f, sf) \in Mand$: if $f \in F$ then $(f, sf) \in R$; (3) if $(f, sf) \in Alt \wedge f \in F$ then $\exists!\, sf' \in \wp(F)\colon sf' \subseteq sf \wedge (f, sf') \in R \wedge \#sf' = 1$; (4) if $(f, sf) \in Or\text{-}rel \wedge f \in F$ then $\exists!\, sf' \in \wp(F)\colon sf' \subseteq sf \wedge (f, sf') \in R \wedge \#sf' \geq 1$.

Definition 3. The *kernel N* of a FM *(Funcs, f_0, Mand, Opt, Alt, Or-rel)* is the set of functionalities, which are present in all configurations, inductively characterized by the following rules: (1) $f_0 \in N$; (2) if $f_1 \in N \wedge (f_1, \{f_2\}) \in Mand$ then $f_2 \in N$.

4. SCs with Variabilities

In this section we extend the SCs with *optional (variant) elements* and later on we establish the binding between these elements with functionalities of a FM, in order to model the behavior of a PL. We will call our proposed extended machines StateCharts* (SCs*).

4.1 Graphical Representation of a SC*

The representation of the optional elements that extend the system kernel in a SC* are depicted in figure 2. We use dashed lines to graphically denote both optional states as well as transitions.

A
Optional State
t : b,c / <β>
Optional Transition

Figure 2. Optional state and Optional transition.

4.2 Abstract Syntax of a SC*

Let *S**, *TR**, *Π**, *A** be set of states, transitions, events and actions of a SC*, respectively. Now the terms that define a state have an additional component $s_{op} \in \{optional, non\_optional\}$ that we will call *StateType(s),* which indicates whether the state *s* is optional or not. Similarly, we add component $t_{op} \in \{optional, non\_optional\}$ to the transitions, and we denote it by *TransType(t)*. We also define the following sets of SC*s optional elements: $SOp \subseteq S^*$, $TOp \subseteq TR^*$ and $VarElem = SOp \cup TOp$.

We will refer to states directly by their names, when these are unique for every state in all the SC*; otherwise, we will use the dot (.) as separator between state and substate names. A transition name is built by the trigger event name followed by source and target state names, respectively.

4.3 Case study: MPs

We considered here a family of MPs which share some functionalities, such as, for example, the capacity of reproducing monophonic sounds and vibration. Optionally, we could incorporate into the kernel of functionalities the capacity to make calls by means of quick-marked, to write text messages, to administer multimedia contents, and combinations of these, such as messages with multimedia content (images, polyphonic sounds and videos).

In order to exhibit an example in the development of this article, we formulate in figure 4 a FM, using the notation proposed by Czarnecki, that relates the involved functionalities in the partially described MP of figure 3.

4.4 Relation between FMs and SCs*

FMs and SCs* are complementary. Both model different aspects of a system and in our proposal, will not be treated independently, since SC* elements model behaviors of present functionalities in the FM. In general, a functionality is described by more than one SC* element. Due to this, we must introduce a relation that binds FM elements with SC* elements.

We define then a function *Imp,* which represents the association between the SC* variant elements and the functionalities of the FM. This way we establish what variant elements of the SC* implement the characteristics of the system described in the FM. Given a FM *(Funcs, f_0, Mand, Opt, Alt, Or-rel)* and a SC* (*S**, *TR**, *Π**, *A**), the type of the function *Imp* is as follows: *Imp*: *Funcs* $\rightarrow$ $\wp$*(VarElem)*, where *VarElem* is the set *SOp*$\cup$*TOp*, *SOp* $\subseteq$ *S** and *TOp* $\subseteq$ *TR**. Sets of variant elements into *Imp* not necessarily must be disjuncts, considering that a state or transition can be part of the implementation of one or more functionalities.

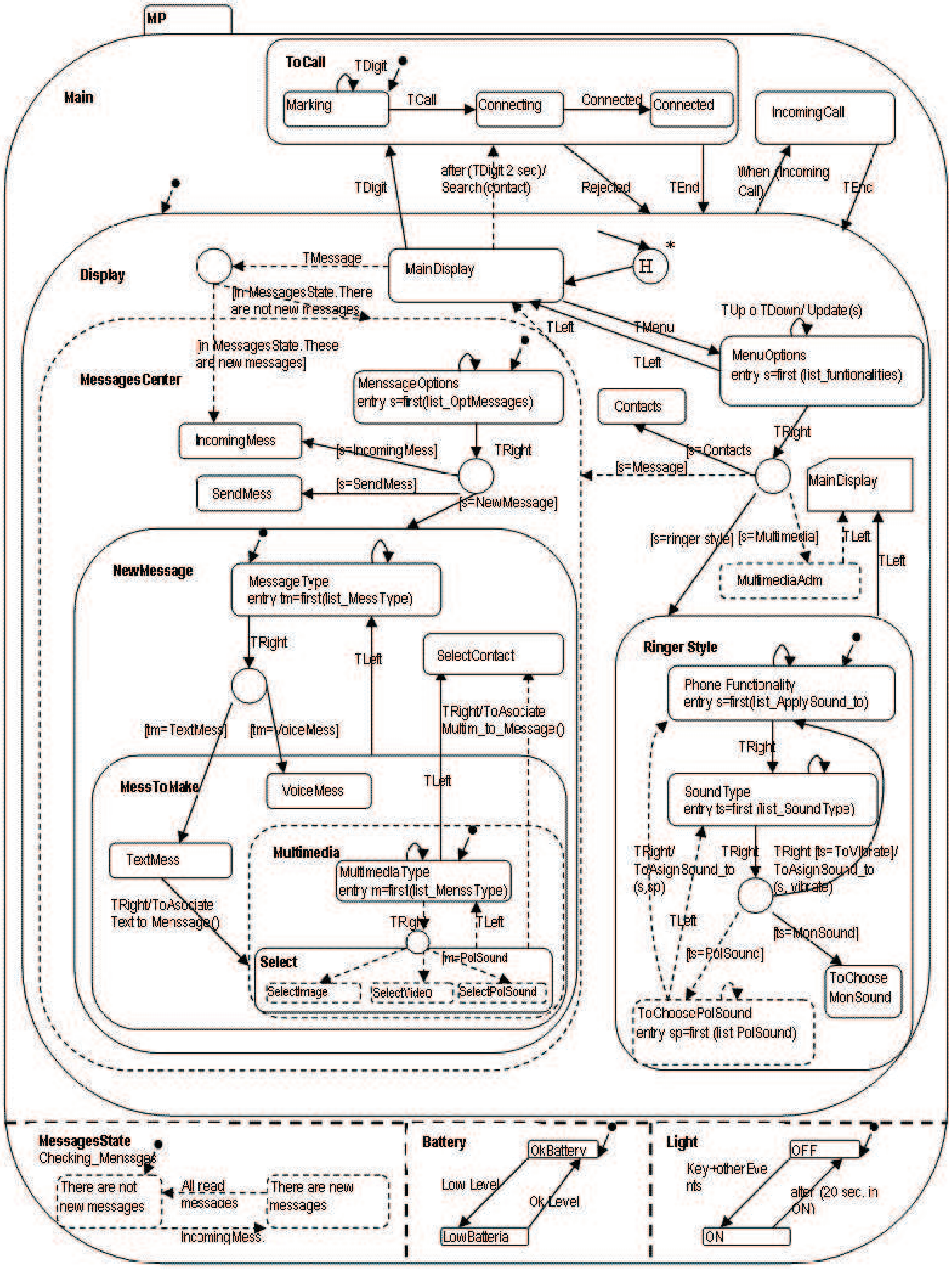

Figure 3. SC* of MPs.

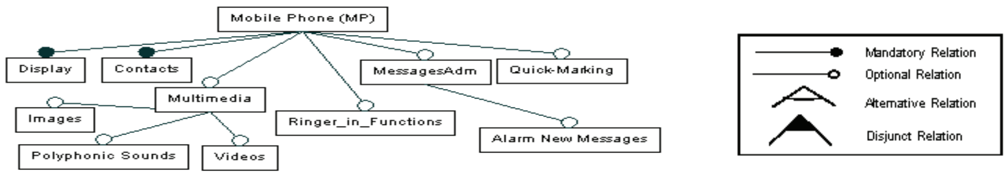

Figure 4. FM of SC* of figure 3.

Taking into account that the mandatory functionalities are always present in all products of the line, it is not necessary to define the SC* syntactic elements that these implement. However, it is necessary to do it for those functionalities that cannot belong to FM configuration. *Imp* will be then a *partial* function defined on FM elements which do not belong to the *kernel*. Therefore, the behavior of a PL is defined by a FM, a SC*, and a function of implementation that binds them.

Example 1. Taking the FM of figure 4 and the SC* of case study of figure 3, we define SC* elements that implements functionalities of MP as follows:

Imp(Polyphonic Sounds) = {SelectPolSound, TRightMultimediaType-SelectPolSound, ToChoosePolSound, TRightSoundType, ToChoosePolSound, TLeftToChoosePolSound-SoundType, TRightToChoosePolSound-Phone Funcionality, ...}; *Imp*(Multimedia) = {Multimedia, AdmMultimedia, TRightrMultimedia.Selecct-SelectContact} ∪ *Imp*(Images) ∪ *Imp*(Polyphonic Sounds) ∪ *Imp*(Videos); *Imp*(MessagesAdm) = {MessagesCenter, TMessage-MainDisplay-IncomingMess, TMessage-MainDisplay-MessagesCenter, TLeft-MessagesCenter-MainDisplay, TRight-OptionsMenu-MessagesCenter} ∪ *Imp*(Alarm New Messages); *Imp*(Alarm New Messages) = {MessagesState, TMessageMainDisplay-IncomingMess}.

5. Instantiation of StateCharts with Variabilities

A FM configuration defines a product or concrete system given a set of selected characteristics. Given a FM configuration and the SC* corresponding to the FM (linked via a function *imp*), we define an *instantiation function* that returns a SC, which specifies the defined product behavior, *Inst: SC* × FMConf → SC.*

We eliminate of the SC* both states and transitions which implement functionalities not present in FM configuration, via the use of the function *Imp* defined in the previous section. The direct elimination of states as well as transitions of the SC* is not trivial. The suppression of SC* components without establishing a control can return inconsistent results, such as, for example, unreachable states or transitions without target. A control and rebuild mechanism of SCs starting from a SC* is defined in such a way that a concrete product is obtained.

In section 5.1 we present the cases and rules of rebuilding that constitute the base of the instantiation method which we included in section 5.2. Later, in section 5.3 we analyze our case study: MPs.

5.1 Cases and Rebuilding Rules

Case 1. When a state is deleted

Case 1.1. When a simple state is deleted

If a simple state *s = [E]* is deleted, then their entry and exit optional transitions are deleted, while the mandatory transitions are composed using the following rebuilding method.

Let $E \in SOp$ be the state to eliminate, $A_1, \ldots, A_n$ predecessor states of *E* (i.e., states from which there are non-optional transitions with target *E* ($t_{AE\_1}, \ldots, t_{AE\_n}$)), and $S_1, \ldots, S_m$ successor states of *E* (i.e., target states of non-optional transitions with source *E* ($t_{ES\_1}, \ldots, t_{ES\_m}$)). When the variant state *E* is deleted, all entry and exit transitions linked to *E* are deleted. Simultaneously, new transitions are generated by the composition of the non-optional entry transitions ($t_{AE\_1}, \ldots, t_{AE\_n}$) with the non-optional exit transitions ($t_{ES\_1}, \ldots, t_{ES\_m}$).

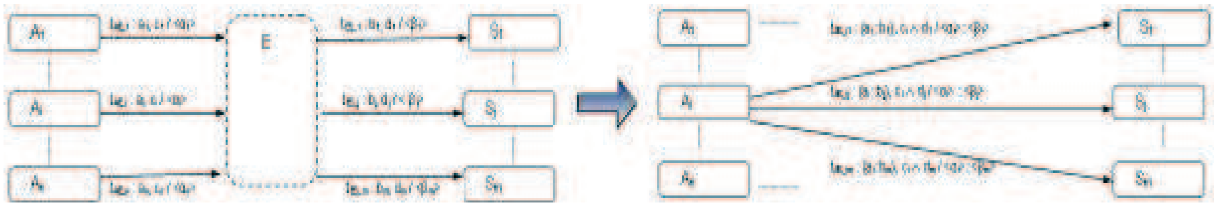

Figure 5. Resulting SC* after the deletion of the optional state.

The composition of two transitions t_1 = (t_1, so_1, e_1, c_1, α_1, sd_1, ht_1, non_optional) and t_2 = (t_2, so_2, e_2, c_2, α_2, sd_2, ht_2, non_optional) define a new transition as follows: comp(t_1, t_2) = (t_{12}, so_1, e_1:: e_2, $c_1 \wedge c_2$, α_1:: a_2, sd_2, ht_2, non_optional), where :: is the sequential composition of events and actions, and $\wedge$ the conjunction of conditions. Both operations must be associative in order to make the instantiation method deterministic.

Let sc = (S*, TR*,Π *, A*) be SC* (in future, we will omit the components Π* and A*), we define the set of entry and exit non-optional transitions pairs of a state E of sc as follows:

$T_{e\_s}(E) = \{(t_e,\ t_s) \in TR^* \times TR^* \mid target(t_e)=E \wedge source(t_s)=E \wedge TransType(t_e)=TransType(t_s)= non\_optional\}$.

The result of eliminating an optional Or-state E of sc corresponds to the following SC*:

Delete_simple_state (E, (S, TR*)) =* $(S^*- \{E\}, TR^* \cup \{ comp(t_e,t_s) \mid (t_e, t_s) \in T_{e\_s}(E) \} - \{ t \in TR^* \mid t \in Domain(T_{e\_s}(E)) \vee t \in Range(T_{e\_s}(E)) \} - \{ t \in TR^* \mid TransType(t)=optional \wedge (source(t)=E \vee target(t)=E) \})$

We call this rule *Delete_simple_state(E, (S*, TR*)).* Figure 5 shows the result of their application.

Case 1.2. When a Or-state is deleted

If an Or-state $s = [E, (s_1 ,..., s_k), l, T]$ is deleted, then their entry and exit optional transitions are deleted, while the mandatory transitions are composed using the following rebuilding method.

The proposal consists in applying the previous transition composition method of case 1.1 on *E*, considering certain conditions and affectations to SC*. Let *sc* = (S*, TR*)* be a SC*, we previously define the set of all the entry and exit non-optional transition pairs of a Or-state *E* of *sc** as follows:

$T_{e\_s}(E) = \{(t_e,t_s) \in TR^* \times TR^* \mid target(t_e) \in sub\_states(E) \wedge source(t_s) \in sub\_states(E) \wedge TransType(t_e)= TransType(t_s)=non\_optional\}$.

We establish that each entry transition to *E* is composed with one exit transition if the source state of the exit transition is reached from the target state of the entry transition. We define *Reachable(E, A)* as the set of reachable substates of *E* from the substate *A*. Formally, we define $TComp_{e\_s}(E) = \{(t_e, t_s) \in T_{e\_s}(E) \mid source(t_s) \in Reachable(E, target(t_e))\}$ as the set of transition pairs that must be composed by means of case 1.1, previous modification of these transitions as is indicated as follows. For each entry transition $t_e \in Domain(TComp_{e\_s}(E))$ its target state is now *E*, i.e., $target(t_e)=E$. Also, for each exit transition $t_s \in Range(TComp_{e\_s}(E))$, $source(t_s)=E$. The result of eliminating the optional Or-state *E* of *sc** corresponds to the SC* following:

Delete_Or_state (E, (S, TR*)) =* $(S^*- (\{E\} \cup sub\_states(E)), TR^* \cup \{ comp(change\_target(t_e,E), change\_source(t_s ,E)) \mid (t_e, t_s) \in TComp_{e\_s}(E) \} - \{ t \in TR^* \mid source(t) \in (\{E\} \cup sub\_states(E)) \vee target(t) \in (\{E\} \cup sub\_states(E)) \})$

change_target(t,E) change the target of transition *t*, such that *target(t)=E*. Likewise, *change_source(t, E)* change the source of transition *t,* such that *source(t)=E.*

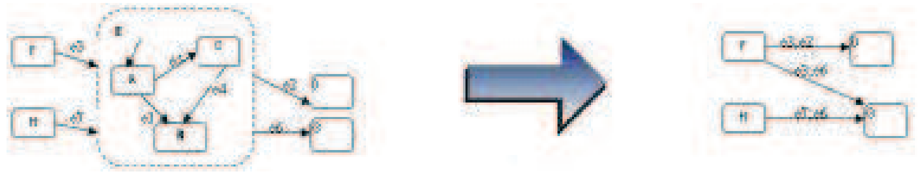

Figura 6. Deletion of an Or-state optional state.

We call this rule *Delete_Or_state(E, (S*, TR*))*.

Case 1.3. When an And-state is deleted

If an optional And-state *E* is deleted, then their entry and exit optional transitions are deleted, while the mandatory transitions are composed using a similar rebuilding method to case 1.2.

Two possible relations of dependency or synchronization between parallel states exist. One of them refers to the occurrence of an event that produces the trigger of two or more transitions belonging to each one of the parallel substates. The second relation corresponds to using conditions of type "*in E*" (see case 3). The latter forces to redefine the concept of reachability, since it is not valid to apply the previous definition of reachability in a way independent in each one of the orthogonal states.

Let *E* be an And-state with *n* orthogonal states. We define now $Reachable(E\,,\,(E_1,\,...,\,E_n\,))$ as the set of n-tuples of reachable states from $(E_1,\,...,\,E_n)$. In this way, maintaining the definition of $T_{e\_s}(e)$ of the previous case and redefining $TComp_{e\_s}(E)$, it is possible to solve the method of transition elimination and composition (*Delete_And_state*) in an analogous form to case 1.2 (see figure 7). $TComp_{e\_s}\,(E) = \{\,(t_e,\,t_s) \in T_{e\_s}(E) \mid (\exists\ n\text{-}tuple\_init,\ n\text{-}tuple\_end \in (S^* \times ... \times S^*)_n \mid\ n\text{-}tuple\_end \in Reachable(E,\ n\text{-}tuple\_init) \wedge (\exists i,j\ 1 \leq i,j \leq n \mid n\text{-}tuple\_end[i] = source(t_s) \wedge n\text{-}tuple\_init[j] = target(t_e) \wedge cond(t_s)\,))\,\}$, being *n* the amount of orthogonal states in *E*.

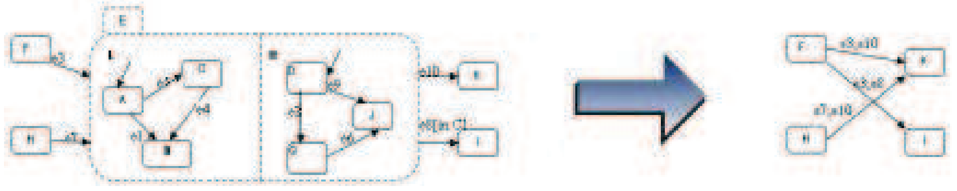

Figure 7. Deletion of an optional And-state.

Note in figure 7 that the condition [in C] in the transition e8 is eliminated (see case 2.2).

We call to this rule *Delete_And_state (E, (S*, TR*))*.

Case 2. Consequences of the elimination of a state

Some situations can appear as a consequence of applying the described cases previously, which must be considered in order to reestablish the SC. These situations are analyzed in the following cases.

Case 2.1. When an initial state is deleted

If an initial state of a state $E = [s,\,(s_1\,,...,\,s_k),\,l,\,T]$ is eliminated, then anyone of their successors that belongs to *E* becomes the new initial substate, i.e. if $\exists t,\ t=(t_{name},\ s_1,\ e,\ c,\ \alpha,\ s_{new\_initial},\ ht) \in T \mid s_{new\_initial} \in (s_2\,,...,\,s_k)$ then $initial'(E) = s_{new\_initial}$.

We call this rule *Delete_Initial_state(E, (S*, TR*))*.

Case 2.2. When some substate in a parallel decomposition is deleted

The conditions of transitions in a parallel decomposition of type "*in E*" are deleted when the state *E* is eliminated via some FM configuration, i.e. if *E* is deleted and $t = (tname,\ s_o,\ e,\ c,\ \alpha,\ s_d,\ ht) \in TR \mid$ *"in E"* $\in c$, then *t' = (tname, s*$_o$*, e, delete_expresion_in_condition("in E",c),* where the function *delete_ expresion_in_condition(expr, c)* eliminates the logic subexpression of *c*.

We call this rule *Delete_condition(E, (S*, TR*))*.

Case 2.3. When all the substates are deleted

If all substates of a superstate *E* disappear, then *E* also disappears, being applied the case 1. The substates can disappear by being involved in implementations of different functionalities.

That is to say, for all Or-state $E = [s, (), l, T] \in S^*$, we apply the rule *Delete_simple_state(E, (S*, TR*))*.

We call this rule *Delete_Or-state_empty (E, (S*, TR*)).*

Case 3. When a transition is deleted

If a transition *t* of a SC* disappears, it does not produce alterations in the SC*, except when some state, or possibly a complete subpart of SC*, is unreachable from the initial state of the system. In this last case, the unreachable substatechart should be eliminated.

We call this rule *Delete_transition(t, (S*, TR*)).*

Case 4. Changing the optional elements to non_optional elements

This rule will be used in the reduction strategy, in order to make part of the final product all optional elements remainders. That is to say, $\forall s \in SOp^*$, *StateType(s') = non_optional*. In a same way, $\forall t \in TR^*$, *TransType(t') = non_optional.*

We call this rule *Changing_optional_to_non_optional(S*, TR*).*

5.2 Instantiation Method

Given a FM and its configuration, we will name *NSF* to the set of *non-selected functionalities* of the model as consequence of the configuration. Formally, for the FM *fm* = (*Funcs,* f_0*, Mand, Opt, Alt, Or-rel)* and a configuration $conf_{fm} = (F, R)$ of *fm, NSF = Funcs - F.* We define also the set of *non-selected components* (*NSC*) by the configuration $conf_{fm}$ of SC* *sc*, which will not be part of the resulting SC, as follows: $NSC(conf_{fm}, sc) = \{ x \in VarElem \mid \exists f_i \in NSF: x \in Imp(f_i) \wedge \neg\exists f_{i'} \in F: f_{i'} \neq f_i \wedge x \in Imp(f_{i'}) \}$, with *VarElem* e *Imp* defined for *sc* according to section 4.4. This is, the states and transitions that do not implement selected functionalities by a configuration will be excluded, through the rules, from the behavior of resulting SC.

The rules should be executed in a certain order, so that be deterministic the obtained SC. We will assume that rules are organized in rule sets, which are then organized in a sequence of rule sets in which each rule set can be considered as a layer [12]. Within a rule set, rules maybe applied in a non-deterministic order. Syntactically we express layers of rule sets as follows: assuming three rules *p*1, *p*2, and *p*3, $\langle \{p1, p2\}, p3\downarrow \rangle$ specifies two layers. The first one containing *p*1, *p*2, and the second containing *p*3. This means that first any rule in the first layer is applied, and then the one in the second layer. The symbol p↓ denotes that the rule *p* is iterated until it cannot be applied anymore.

The order of application of the rules will determine the order of selection of the elements in $NSC(conf_{fm}, sc)$. Let $sc \in SC^*$, $fmc \in FMConf$, and $E,t \in NSC(fmc, sc)$. We establish the following application order of rules:

> ⟨ {*Delete_Initial_state(E, sc), Delete_condition(E, sc)*}↓; *Delete_simple_state(E, sc)* ↓; {*Delete_ Or_state(E, sc)*, *Delete_And_state(E, sc)*}↓; {*Delete_Or-state_empty(E, sc), Delete_transition(t, sc), Changing_optional_to_non_optional(sc)*} ⟩

It is important to consider that the application of a rule produces the elimination of the elements in *NSC(fmc, sc).* The strategy is completed when $NSC(fmc, sc) = \emptyset$.

The method is deterministic with respect to the resulting SC, and avoid inconsistencies that could arise if the compound states are removed before that the simple states. Note that the implementation of the first two rules not eliminate states, but only change initial states and deleting conditions like "in E". Later the affected states will be eliminated by the successor rules. The rule *Changing_optional_ to_non_optional(sc)* convert optional elements, which remain in *sc*, to non_optional elements.

Termination and Deterministic Implementation

It is important to realize that termination and confluence problems can occur whenever a rule-based approach is used. As termination and confluence are fundamental for the correctness of a model transformation, systematically validating these properties is a main prerequisite for successful practical applications, such as transformations in the MDA context, for example.

In our context, the property of termination is clear, given that rules are applied on elements of *NSC(fmc, sc)*, and this are eliminated in each application until *NSC(fmc, sc)=∅*. The determinism is analyzed below.

The rol of the first two rules is to avoid ill-formed statecharts. The main rule is *Delete_simple_state(E, sc),* since *Delete_Or_state(E, sc)* and *Delete_And_state(E, sc)* are based on this. Note that *Delete_simple_state(E, sc)* not produce ill-formed statecharts (see case 1.1).

Let *E1* and *E2* $\in$ *NSC(fmc, sc)* be the states to eliminate. We consider both independent if there is no transition connecting both, in this case is clear that the elimination in any order produces the same SC. Figure 8 shows the opposite case, which is discussed below.

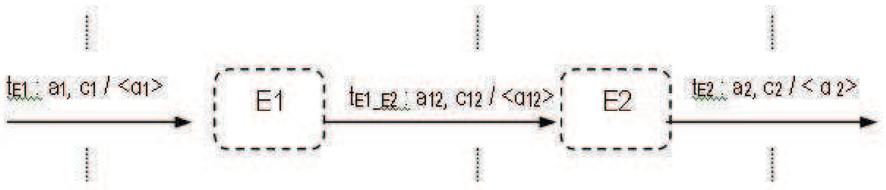

Figure 8. Deterministic elimination.

Let $t_{E1} = (t_{E1}, so_1, a_1, c_1, \alpha_1, E1, ht_1, non\_optional)$, $t_{E1\_E2} = (t_{E1\_E2}, E1, a_{12}, c_{12}, \alpha_{12}, E12, ht_{12}, non\_optional)$ and $t_{E2} = (t_{E2}, E2 , a_2, c_2, \alpha_2, sd_2, ht_2, non\_optional)$ be the transitions as shown in figure 8.

We assume that the rule *Delete_simple_state(E1, sc)* is applied before that the rule *Delete_simple_state (E2, sc)*. According to the definition of *Delete_simple_state(E1, sc)*, are composed all input non_optional transitions with output non_optional transitions of *E1*. In particular, t_{E1} is composed with t_{E1_E2}.

$comp(t_{E1}, t_{E1\_E2}) = t_{E1\_(E1\_E2)} = (t_{E1\_(E1\_E2)}, so_1, a_1 :: a_{12}, c_1 \wedge c_{12}, \alpha_1 :: \alpha_{12} , E2, ht_{12}, non\_optional)$

We apply now the rule *Delete_simple_state(E2, sc)* compose transitions $t_{E1\_(E1\_E2)}$ with t_{E2},

$comp(t_{E1\_(E1\_E2)}, t_{E2}) = t_{E1\_(E1\_E2)\_E2} = (t_{E1\_(E1\_E2)\_E2}, so_1, (a_1 :: a_{12}) :: a_2, (c_1 \wedge c_{12}) \wedge c_2, (\alpha_1 :: \alpha_{12}) :: \alpha_2 , sd_2 , ht_2, non\_optional)$

On the other hand, if we assume now the inverse application of both rules, we can see that the resulting composition has the same result as the previous case.

$comp(t_{E1}, comp(t_{E1\_E2}, t_{E2})) = t_{E1\_(E1\_E2)\_E2} = (t_{E1\_(E1\_E2)\_E2}, so_1, a_1 :: (a_{12} :: a_2), c_1 \wedge (c_{12} \wedge c_2), \alpha_1 :: (\alpha_{12} :: \alpha_2) , sd_2 , ht_2, non\_optional)$

Given that the operators $\wedge$ and $::$ are associative, $comp(comp(t_{E1}, t_{E1\_E2}) , t_{E2}) = comp(t_{E1}, comp(t_{E1\_E2}, t_{E2}))$.

The rules *Delete_Or_state* and *Delete_And_state*, based on the rule *Delete_simple_state*, previously filters input and output transitions which will be composed in the same way as in *Delete_simple_state*. Therefore we may consider to *E1* and *E2* of previous case, as compound states.

5.3 Instantiation of the Case study: MPs

The FM of figure 4 can be configured to characterize different MPs, according to the specification of the case study of section 4.3. Next, we present a configuration of a MP of figure 4 and we proceed towards obtaining the corresponding SC (the MP wanted), according to the application order of rules defined in section 5.2.

A MP with neither the support for the management of polyphonic sounds nor the capacity of alerting the user when new messages enter in the incoming mailbox, is defined by the configuration $conf_{fm}$ = *(F, R)* of the FM of figure 5, where:

F = {MP, Display, Contacts, MessagesAdm, Multimedia, Images, Videos, Quick-Marking, Ringer_in_functions}, and

R = {(MP, {Multimedia}), (MP, {MessagesAdm}), (MP, {Quick-Marking}), (MP, {Display}), (MP, {Contacts}), (MP, { Ringer_in_functions}), (Multimedia, {Images}), (Multimedia, {Videos})}.

Taking the previous configuration and the function *Imp* described in example 1 of section 4.4, the sets *NSF* and *NSC*($conf_{fm}$*, sc)* are defined as follows:

NSF = {Polyphonic Sounds, AlamrNewMessages}, and

NSC($conf_{fm}$*, sc) = {SelectPolSound, TRightMultimedia Type-SelectPolSound, ToChoosePolSound, TRightSoundType-ToChoosePolSound,TLeftToChoosePolSound-SoundType, TRightToChoosePol Sound-PhoneFuncionality, MessagesState, TMessage-MainDisplay-Incoming Mess}.*

None component of *NSC*($conf_{fm}$*, sc)* is an initial state, so it is not possible to apply the rule *Delete_Initial_state*. However, it is possible to apply the rule *Delete_condition (MessagesState, sc)*, since that remove all type conditions *"in MessagesState"*, but does not eliminate the state *MessagesState*. The case study shows the occurrence of the condition "*in MessagesState*" on the transition *TMessage*. In accordance with the established order, in section 5.2, must apply the rules in the following order:

Delete_condition (MessagesState, sc) ➔
Delete_simple_state(SelectPolSound, sc) ➔
Delete_simple_state(ToChoosePolSound, sc) ➔
Delete_Or-state_empty(MessagesState, sc) ➔
Delete_transition(TRightMultimedia Type-SelectPolSound , sc) ➔
Delete_transition(TRightSoundType-ToChoosePolSound, sc) ➔
Delete_transition(TLeftToChoosePolSound-SoundType, sc) ➔
Delete_transition(TRightToChoosePolSound-PhoneFuncionality, sc) ➔
Delete_transition(TMessage-MainDisplay-Incoming Mess, sc) ➔
Changing_optional_to_non_optional(sc).

The resulting SC is observed in figure 9.

6. Related Works

A variety of existing approaches propose the incorporation of variability in software systems and in particular on PLs. One of these is the one designed by Jacobson [15], whose weaknesses have been analyzed by several authors.

Most recent solutions attempt to modify or to extend the used models. Different authors have proposed to represent explicitly the variation points by adding annotations or by changing the bases of Use Case Diagrams. For example, Von der Maßen [4] proposes the use of a graphical notation with new relations. John and Muthig [16] suggest the application of use case templates, although they do not distinguish between optional, alternative or obligatory variants. However, Halman and Pohl propose in [17] to make use of UML 2 package merge, based on [6], as a tool for the variability representation and configuration in PLs. As opposed to the mentioned proposals, our solution is centered in one behavior specification model of a PL, as are the SCs, introducing a clearly defined formal sustenance.

Also, to define PLs and to characterize their different products we use FMs, which admit a formal definition and allow us to configure the functional characteristics of a line. An alternative approach to this paper is developed in parallel by other members of the research project in which this work is subsumed. In [18] the authors, in a formal framework, define functions that associate SCs (not components of SCs, as in our case) to functionalities of a FM, and analyze forms of combination between different SCs which specify possible variants of a PL. Whereas under our method the behavior of a product into a PL is obtained basically by a selection process, in [18] the focus is oriented towards a process of SC combinations.

Finally, a preliminary version of this work is [11]. As opposed to [11], this paper presents a rule-based approach. We define an application strategy for the rules in a proper manner, in such a way that inconsistencies are avoided in the statechart obtained. We also formalized and added rules that in [11] are omitted or only described informally.

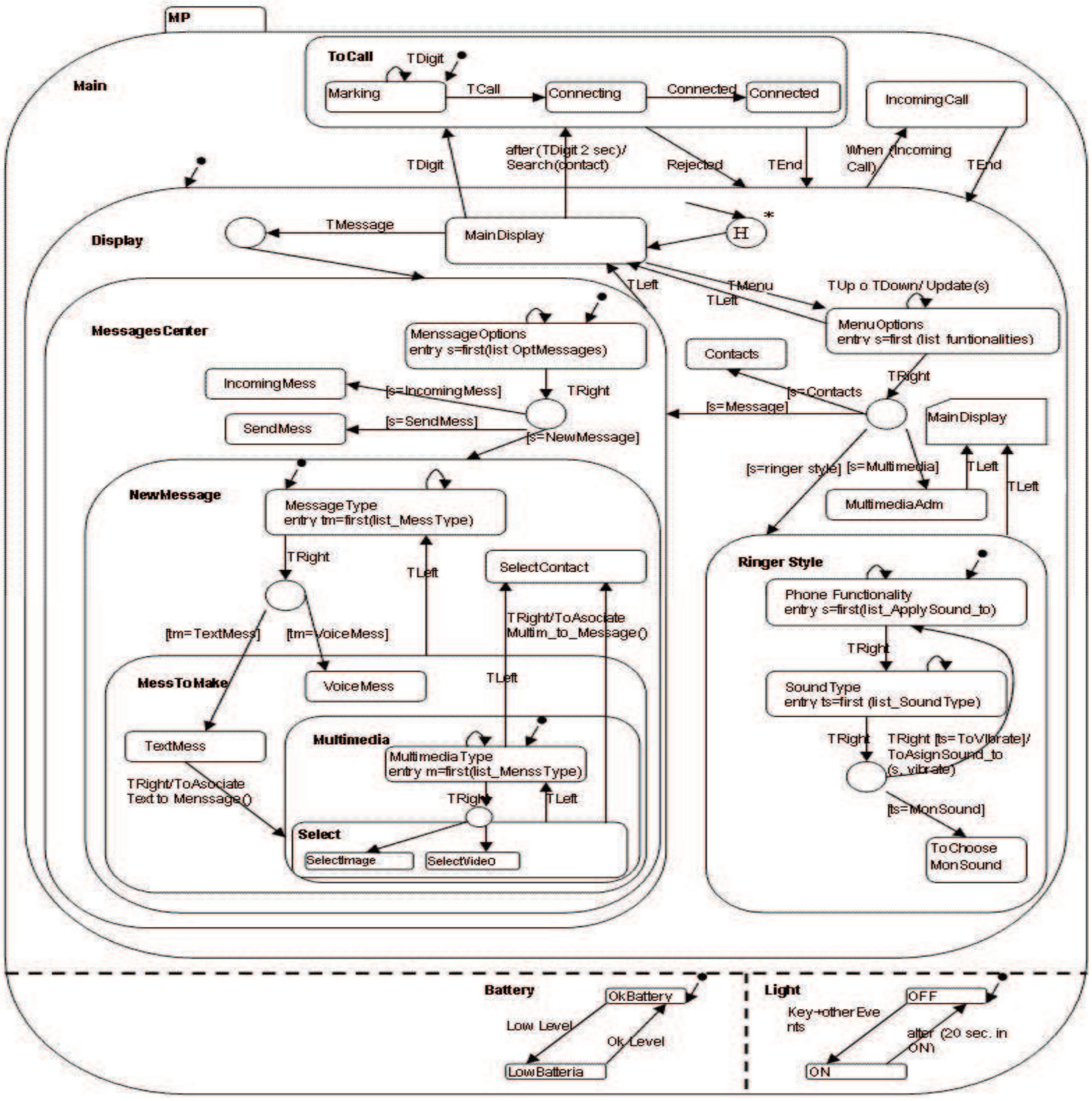

Figure 9. SC for a MP without polyphonic sounds and without alert of new messages.

7. Conclusion and Further Works

Most of the techniques of Model Driven Development make use of UML. In particular, the SCs of UML constitutes a mechanism for specifying systems behavior by means of a graphical representation. In this work, we presented an extension of UML SCs by incorporating variability in their essential components to specify PLs. The variability is introduced in the SCs distinguishing optional and non-optional states as well as optional and non-optional transitions. A PL is specified with a SC*, a FM, and a formal relation (an implementation function) that binds both models. Using FMs to describe the common and variant functionalities and applying a rule-based instantiation method, concrete SCs corresponding to different PLs can be obtained. The approach defines the transformation strategy from extended SCs to standard UML SCs. We develop partial examples of a case study based on mobile phone technology, whose full version is not included in this article due to space restrictions.

Given the fact that UML and SCs have become very successful languages for analysis and design in the very short run, we are confident that the results of this work can be successfully applied to the real problems of the software industry. It is a timely contribution to an authentic and actual problem.

As part of our plans for future work, we are interested in an extension of SCs which allows us to completely cover the UML 2.0 SCs and analyze variabilities, not only the ones considered in this paper, but in all of their components. Also, we will make an attempt to provide a formal semantics for the extension. This semantics is an essential preliminary step towards both the automatic code generation and the validation of complex software systems. Finally, we will try to compare formally both proposed methods by members of the investigation project in which this work is subsumed, in order to complement and enrich both lines of research.

References

[1] P. Clements, L. Northrop. Software Product Lines: Practices and Patterns. SEI Series in SE, Addison-Wesley. 2002.

[2] H. Gomaa. Designing Software Product Lines with UML. The Addison-Wesley Object Technology Series, 2004.

[3] K. Kang, J. Lee, P. Donohoe. Feature–Oriented Product Line Engineering. IEEE Software, 19(4):58–65, 2002.

[4] T. von der Maßen, H. Lichter. Modeling Variability by UML Use Case Diagrams. In proc. of REPL'02, 2002.

[5] G. Halmans, K. Pohl. Communicating the Variability of a Software-Product Family to Customers. Journal of Software and Systems Modeling, 2(1):15-36, 2003.

[6] A. Zito, Z. Diskin, J. Dingel. Package merge in UML 2: Practice vs. theory?. In Proc. of MoDELS/UML 2006, October 2006.

[7] M. Clauß. Generic modeling using Uml extensions for variability. In Workshop on Domain Specific Visual Languages at OOPSLA 01, 2001.

[8] K. Czarnecki, M. Antkiewicz. Mapping Features to models: a template approach based on superimposed variants. In proc. of GPCE'05, LNCS 3676, Springer, pp. 422-437, 2006.

[9] Object Management Group: OMG Unified Modeling Language Specification Version 2.0, 2004.

[10] K. Czarnecki, U. Eisenecker. Generative Programming: Methods, Techniques, and Applications. Addison-Wesley, 2000.

[11] C. Luna, A. Gonzalez. Behavior Specification of Product Lines via Feature Models and UML Statecharts with Variabilities. In proc. of the XXVII International Conference of the Chilean Computer Science Society, Chile, IEEE CS Press. pp. 32-41, 2008.

[12] Jochen M. Küster. Definition and validation of model transformations. Journal of Software and Systems Modeling, 5(3) 233-259, 2006.

[13] D. Harel: Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8: 231-274,1987.

[14] M. von der Beeck. A structured operational semantics for UML-statecharts. Springer, 2002.

[15] I. Jacobson et al. Software Reuse. Architecture, Process and Organization for Business Success. Addison-Wesley, 1997.

[16] I. John, D. Muthig: Tailoring Use Cases for Product Line Modeling. In Proc. of the International Workshop on Requirements Engineering for Product Lines 2002 (REPL'02), 2002.

[17] M. Laguna et al. Gestión de la Variabilidad en Líneas de Productos. In Proc. Of CLEI´07, Costa Rica, Octubre de 2007.

[18] N. Szasz, P. Vilanova. Statecharts and Variabilities. In Proc. of Second International Workshop on Variability Modelling of Software-intensive Systems, Essen, Germany, January 2008.