Variability Management on Behavioral Models

Patrick Tessier, David Servat, Sébastien Gérard
CEA, LIST, Gif-sur-Yvette, F-91191, France

Abstract

This paper deals with managing variability on behavioral models. Such models are generally more complex, less tractable by hand than the static, structural parts of a system description. This calls for specific support to check the consistency of variability expression model-wide: defining some elements as variable may impact several behavioral constructs, even elements that designers may not be aware of – e.g. events attached to triggers. Moreover, variable elements at the structural level usually imply variable behavioral constructs in ways that are not easily foreseeable: a seemingly perfectly valid variability scheme may lead to ill-formed behavioral models after derivation. This paper takes UML state machine diagrams as a case study and presents some technical solutions to maintain consistency between both levels: a propagation mechanism to deal with the impact of variability model-wide and a constructive method to check the well-formedness of state machines obtained by derivation.

1. Introduction

The system family paradigm was first proposed in [13, 15]. It has been the research topic of several research projects such as ESAPS, CAFÉ, Families\. Their focus was to foster reuse of models for a whole set of similar applications. An application is seen as a specific instance of an application family called an application domain. This is known as the “product line” approach [4]. A system family model relies on the design and composition of common and specific functionalities, called variable elements. Each specific member of a system family, called the product model, results from one derivation of one single system family model.

The purpose of our work is to apply the system family paradigm to the development of real-time systems. Our research team proposes extensions and design method guidelines to help engineers improve their experience of the UML in this particular domain.

We participated in the recent advent of the MARTE profile, which is the standardized UML profile for real-time and embedded systems design and analysis [14]. For such systems, behavioral models are critical and complex whereas structural models remain fairly easy to understand and manage. Structural and behavioral parts of a system description usually present complex links that are hard to manage when one wants to depict commonalities and differences between models. Designing families of real-time systems is not an easy task and calls for specific support: 1) means of expressing variability; 2) consistency checks to help manage variability model-wide, both at structural and behavioral levels; 3) correct-by-construction derivation of product models from the system family model.

This paper presents some results of our ongoing work in this context within the limited scope of models where behavior is depicted with a set of state machines attached to structural elements. After a brief overview of related works, the paper is organized into three sections: the first one (section 3) presents the chosen conceptual model and UML profile used to express variability on models. A case study is introduced at the end of this section. Then we describe two specific mechanisms that help designers maintain consistency on their variability-enabled models: 1) a mechanism of variability propagation ensures that the impact of the addition of variability on any modeling elements is fully reflected throughout the entire model (section 4); 2) a mechanism to check the well-formedness of state machines obtained by derivation is described in section 5. Both of these mechanisms help monitor the derivation process of a product model from the system family model. In the course of this paper we show how this in turn helps assess the consistence and completeness of the system family model as a whole.

2. Related works and overall context

Means of expressing variability have been the subject of many research works. Those grounded in the UML language have usually pointed out the lacks of UML2 (see for instance [16]) and extensions have been proposed in terms of profile definitions. Roughly

1 http://www.esi.es/Families/
speaking, the works in this area have consisted in proposing adaptations of established product-line approaches in the UML domain. In [6, 7, 10] the FODA approach [9] was adapted to UML. In [18, 19], ideas from FODA are reused. Results from the Families project led to a conceptual model and a UML profile where several approaches were combined [2].

It is not the goal of this paper to discuss the expressiveness of the various approaches found in this area. They all rely on the distinction between common and variable elements - the latter being usually tagged by some given stereotype - and the addition of constrained relationships between such variable elements. Allowed constraints and choice of constraint language may vary. A rationale is usually provided for the choice of one element among a set of possibilities.

Then the overall process usually follows the same principles from one approach to the other. From a common system family model one intends to drive the generation of several product models, using a decision model to guide the process: the decision model is a graph or tree-based model that lists the various possibilities in terms of variability resolution. Following paths in this graph results in resolving sets of variable elements and progressing towards a model with less variability than before. Ultimately a model with no remaining variable elements is obtained: a product model. The list of choices made in the process characterizes the product in terms of functionalities, qualities, or the like.

Let us make some comments:

1. We may note that for now there is no global convergence towards a standardized UML profile to express variability. The MARTE profile does not fill the gap so far on this aspect.

2. A commonality of these approaches - which is the driver for the present contribution - is that the variability is expressed essentially in structural models. When some approaches describe variability in the behavior, then usually nothing ensures that derived product models are valid. For instance no checks are performed on the derived state machines or behavioral constructs. We think that a more rigorous approach is needed for real-time systems. Variability must be checked so that all possible derivation from a common family model shall produce an acceptable, meaningful product model, e.g. in our case, state machines obtained by derivation must be verified.

3. As said previously, the derivation process usually relies on a guide called decision model or feature model. This model is hand-made during the construction of the system family. This hand-made approach is not tractable when one addresses behavioral models. The presence or absence of structural elements and rationale thereof can usually be traced back to some intelligible needs from a designer point of view. Usually variability in the functional features or the platform choices is at stake. Yet understanding the implication of variability on the topology of transitions and triggers on state machine diagrams is far less intuitive. Even to check that a state machine is well or ill formed is not an easy task to perform manually.

Given this context and bearing in mind the overall complexity of the problems mentioned here, the following sections do not pretend to propose definite solutions but rather some operational mechanisms to help designers maintain consistency within a deliberately limited and constrained set of variability-enabled models. The main characteristic of our approach is to find ways to monitor the derivation process so that the overall consistency and completeness of the system family model can be assessed.

3. Modeling variability in UML

This section presents the meta-model that has been designed to express variability in a UML model. This work is grounded in the proposed conceptual variability model of the Families project [2]. It currently serves as the basis for the proposal of artifact-variability (i.e. design elements centered) of the ATESST\(^2\) project [5] whose goal is to propose a refined version of the EAST-ADL language for automotive systems. The presentation is deliberately limited here to the core elements that enable to understand the UML profile constructs used later in the case study. An extensive description of the conceptual model and profile can be found in [17].

3.1. Meta-model to express variability

A System Family Model (SFM) factorizes several product models into one. It is therefore made up of common elements and variable elements. Model elements not tagged as variable elements are implicitly considered common to all products.

Two kinds of variable elements exist (see figure 1):

- **Variable element**: these are the variable elements that are explicitly introduced by the user.
- **PropagatedVariableElement**: these are elements that acquire the variability feature through their relationships to other possibly variable elements.

\(^2\)http://www.atesst.org/
This is intended to be automatically assigned by a tool.

In our approach, variable elements are propagated from source variable elements along model and meta-model relationships so that impacts of the addition of variability are fully covered. The distinction between both types of variable elements helps to achieve traceability on what the propagation tool produced.

Specifying that a model element is variable is not sufficient to describe a system family model. In fact variable elements are generally not isolated, but constrain one another: presence or absence of one implies various choices on others, etc. One clearly needs to express such dependency constraints between variable elements.

To this end, several approaches propose mechanisms to add constraints between variations. In [3] a “requires” dependency is used to link two variable elements; in [19] OCL constraints are used to define more elaborated variable element dependencies.

In our approach, OCL constraints are also used. Yet because constraints in this formalism may be complex to write, we introduced the concept of a variation group, which features several predefined types of constraints among a set of variable elements: the constraints may be expressed via OCL or with some textual user-defined language.

A variation group features six predefined kinds of variability constraints:

- **Equivalence**: the listed variable elements work as a group, they will be either all present or absent in any product model.
- **Alternative**: only one of the listed variable elements will appear in any product model.
- **OneAmongSeveral**: several variable elements listed – at least one in any case - will be present in the product model.
- **Implication**: a variable element implies the existence of another variable element in the system model.
- **CustomizedCombination**: the constraint between listed variable elements is written directly by the designer. The constraint is a logical expression where operators are not, and, or, xor, implies.

A rationale is provided along with all VariationGroups in the Motivation property that describes the motivation.

These predefined constraint kinds cover most of the common needs, such as the «requires» or «excludes» relationships from the FODA methodology. However at times more complex constraints need to be expressed. For this, the ComplexVariationGroup can be used: it allows specifying constrained combinations of variation groups in a hierarchical manner.

![Fig.1: Meta-model for variability modeling.](image)

### 3.2. A UML profile for variability modeling

This paper is focused on the impact of variability on state machine diagrams. Consequently, the presentation of the profile is limited to what is relevant for state machine diagrams and class diagrams.

A variable element is marked by the stereotype «VariableElement» (Table 1). This stereotype can be applied on a class, a property (an attribute of a Class), or an operation. To express variability in a state machine, the stereotype can be applied on a Transition or a Vertex (a generalized class for PseudoState and State in the UML). Other more elaborated elements of state machine diagrams, such as Entry/Exit points, Regions, ConnectionPointReferences or Ports of protocol StateMachines are not considered for the moment in this work.

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>BaseClass</th>
<th>Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>«VariableElement»</td>
<td>Class</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Property</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Operation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vertex</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transition</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: «VariableElement» stereotype

As said previously, variable elements resulting from propagation of variability along the model and meta model relationships are covered by the «PropagatedVariableElement» stereotype, which
owns a property called VariationOrigin that references the source variation element from which it originates.

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>BaseClass</th>
<th>Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>«PropagatedVariableElement»</td>
<td>Class</td>
<td>VariationOrigin</td>
</tr>
</tbody>
</table>

Table 2: «PropagatedVariableElement» stereotype
Constrained clusters of variable elements are supported by a class stereotyped as “VariationGroup” (table 4). The type of the embedded constraint is specified in the variationGroupKind property. A rationale for the cluster may be specified by the user with the motivation property. A variation group saves references to the clustered variation element via its property variationElements.

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>BaseClass</th>
<th>Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>«VariationGroup»</td>
<td>Class</td>
<td>variationGroupKind</td>
</tr>
</tbody>
</table>

Table 4: “VariationGroup” stereotype

Table 5: Properties of «VariationGroup»

The modeler first defines the elements that are variable by tagging them with the “VariationElement” stereotype. Then in order to introduce constraints between variable elements, he introduces classes stereotyped as “VariationGroup”, fills in the references to the variable elements, provides a rationale and a constraint, for instance in OCL or textual language. The name of the VariationGroup classes does not matter, yet a good practice is to give names relevant to the type of the embedded constraint. A practical example is given in the next section.

3.3. Watch case study
We consider the case of a watch that offers various alarm modes: a sound signal, a visual signal or a combination of both. There are several ways to model such a system. We assume here that the watch system is modeled by a single Watch class (figure 2). It implements two interfaces. The WatchControl interface defines start() and stop() operations to trigger the watch system as a whole. The AlarmControl interface provides operations to trigger the alarm function – startAlarm() and stopAlarm() operations. The Watch class has associations to other elements, namely a Display used for the watch as a whole, a DisplayAlarm specific to the alarm function and a Beeper that provides for the sound mode. To deal with both alarm modes the Watch class features two internal operations, startSoundAlarm() and startVisualAlarm() to trigger the start() operation of either of the associated Beeper or DisplayAlarm. The execution sequence and respective delegation of calls are depicted in the state machine diagram of figure 3.

Consider now that one wants to derive a system family out of this rather complete watch, to factorize other less advanced models – providing either one of the alarm modes or perhaps even featuring no alarm function at all. We assume that the overall system does not change: less advanced systems can be represented by downgraded Watch classes featuring fewer operations and updated execution behavior.

To do so one tags some of the modeling elements with the “VariableElement” stereotype (figure 4):
• *startAlarm()* and *stopAlarm()* should be tagged as variable, because these operations are specific to the alarm functionality.

• *startSoundAlarm()* and *startVisualAlarm()* should also be tagged as variable, because they are only used when the alarm function is enabled.

This ends the presentation of the expression means. We end up with a model where variability has been set on the structural part of the model. In the following section we will see how variability is kept consistent in both behavioral and structural levels and how the derivation process can be checked. The same case study will be considered throughout the paper.

**4. Variability propagation**

Ensure consistency in a system family model is crucial. A lack of consistency may result in ill-formed state machines at derivation time. When a transition is tagged as a variable element, it may be the case that one of the derived state machines features non-reachable states, i.e. with no incoming transition, even though the state itself was not tagged variable. Such states should not be generated.

To cope with that, our approach propagates variation across the system family model, along both model and meta-model relationships. A set of rules has been defined to address various situations. The following are examples of such rules, to cope with Triggers and CallEvents – elements most of the time unknown to users, who are only aware of the Transitions and do not have an in-depth view of the model repository. Rules are followed by their OCL expressions:

An operation specified as a variable element implies that all CallEvents referencing this operation are variable elements and all
Triggers that are associated to this operation via the CallEvents are also variable elements.

Context Trigger inv:
self.event.isKindOf(CallEvent) and
self.event.operation.isStereotyped('VariableElement')
implies
(self.event.isStereotyped('PropagatedVariableElement') and
self.isStereotyped('PropagatedVariableElement'))

If Trigger is a variable element then all associated Transition elements are variable elements.

Context Transition inv:
self.trigger.isStereotyped('VariableElement') implies
self.isStereotyped('PropagatedVariableElement')

If a state is a variable element then all its incoming and outgoing Transitions are variable elements.

Context State inv:
self.isStereotyped('VariableElement') implies
self.incoming
->forAll(t,t.isStereotyped('PropagatedVariableElement')) and
self.outgoing
->forAll(t,t.isStereotyped('PropagatedVariableElement'))

Such a propagation process targets to faithfully reflect the potential impacts of variability on one model entity onto the rest of the model, so as to prevent errors coming from careless addition of variability. Yet on simple cases, it can even be more productive. In our case study, this simple propagation mechanism automatically provides an update of the state-machine diagram from figure 3 according to the variability information put on the structural diagram (figures 4 and 5). The resulting state machine is depicted on figure 6.

Fig. 6: State machine after propagation

This illustrates the interest of such a propagation mechanism, which can even produce behavioral diagrams out of a variability-enabled structural model.

Let us note here that the propagation mechanism does not consider the actual constraints embedded within VariationGroups. Rules are based on containment or cross-reference relationships, not on the semantics of the link between variable elements. Finally it is likely that the modeler shall introduce a range limit to this propagation mechanism, otherwise some rules may result in all model elements being tagged as variables.

At this stage one is assured that the VariableElement stereotype is applied consistently system-wide. The constraints embedded in the VariationGroups can be used to derive what is called a decision model, in which paths represent possible sequence of choices on the variable elements. The construction of the decision model out of the VariationGroup information can be found in [17].

It is however another problem to know whether all paths result in correct product models. If constraints have not been well designed, ill-formed models may be derived. Imagine that the incoming transitions of a state are all variable elements and there exists a decision where all of them are removed from the model, we may end up with a state with no incoming transition, hence an ill-formed state machine.

For example in our case study, the state machine depicted on figure 7 may be derived: it is ill-formed because both states ReadyWithAlarm and RunningWithAlarm cannot be reached.

Fig. 7: Ill-formed state machine

This situation may occur even if such a state was tagged as variable by the propagation mechanism: this process is blind to the constraints embedded in VariationGroups. Such an interesting feature is for the moment beyond the capabilities of our approach. For the state to be removed automatically when no incoming transitions is left, one would have to alter the VariationGroups introduced so that they incorporate the state as part of their managed variable elements. It remains to be investigated whether such an enhanced propagation mechanism would suffice to induce
correct-by-construction derived models. We may have doubts considering the complexity of variability schemes usually imposed on structural models. Thus, one has probably to face the fact that the derivation process may result in ill-formed models. The following section presents means to cope with that, in the case of state machine diagrams.

5. Evaluation of derived state machines

As said previously, the derivation of ill-formed product models from a common system family model is very difficult to avoid. Thus the correctness of derived product model shall be evaluated. In our case, we have to assess whether all possible choices made during derivation result in valid state machines.

To do so, our approach constructs a function that captures the state machine topology. The configuration of its transitions and states are seen as the variables of this function. The evaluation of the function returns $null$ if the state and transition configuration is invalid, else returns a compact representation of the valid state machine.

Based on this, we evaluate the various state machine configurations obtained from each path of the decision model. This helps us evaluate the overall consistency of the system family model. If some branches lead to ill-formed behavioral diagrams then there must be a design fault somewhere in the variability model.

Our study is conducted on state machine diagrams. In this part, formal foundation in briefly explained and then the process is presented. The process is divided in three parts. First the transformation of a state-machine into regular expression is presented. Then the analysis of this expression and finally a process of derivation are described.

5.1. A Formal foundation for derivation

Our approach makes use of the following works on regular expressions, automata and Kleene algebra [8, 11], which are very well suited to analyze automata-based specification and study the impact of variations.

We construct a regular expression that represents a complete state machine, featuring variable and non-variable elements. We then evaluate this expression according to various valuations of its variable elements. The evaluation is made in the Kleene algebra. If the variable element – transition or state – is present, the valuation amounts to the identifier of the element – e.g. transition “t9”, name of state, etc. If the variable element is absent, it amounts to null ($Ø$ in Kleene algebra). As a result, if the overall regular expression evaluates to $null$, the state-machine is ill-formed; otherwise the result represents the topology of the derived state-machine obtained for a given combination of variable elements.

The following paragraphs explain the calculus of the regular expression, its evaluation and analysis.

5.2. A regular expression for state machine

The regular expression should capture the topology of the state-machine. To do that, the UML state machine is first transformed into a regular automaton, defined as follows:

- its alphabet is chosen as the set of UML transition and state identifiers
- its recognized language is chosen as the sequences of transitions and states of the state machine.

Figure 8 shows the corresponding automaton of the complete state machine of our watch case study from figure 6. The regular expression is then computed out of this automaton using a classical algorithm from the literature [12]. The expression is not shown here because of its length.

Fig. 8: Corresponding automaton

5.3. Evaluation of the regular expression

To identify variability impact, the regular expression is considered as a function whose variables are issued from UML variable elements. Each evaluation of this function represents the topology of the state machine after choices on variable elements have been performed.

Variables of the function are easily deduced from the UML variable elements. In our example the variable are $t2$, $t3$, $t6$, $t7$, etc. All combinations of values are not possible for each variable. Indeed, the set of possible value combinations have to be calculated by taking into account the constraints introduced in the VariationGroups. For example, transitions $t2$ and $t6$ always have the same value. The existence of $t2$ and $t6$ depends on the existence of the same variable element, the $startAlarm$ trigger. If this trigger is not present in the product model, both transitions are removed.

The evaluation of the regular expression uses the Kleene algebra. More precisely, the following properties are used:

![Diagram](image)
Let A be an alphabet, and let $L \in P(A^*)$ (all languages based on $A^*$). Then the following properties hold:

\[
\emptyset \cdot L = L \cdot \emptyset = L \\
\emptyset = L \cdot \emptyset = \emptyset
\]

For example, if $t_2, t_3, t_6, t_7$ are evaluated to $\emptyset$ (imagine a decision which gets rid of these transitions), the evaluation of the function is:

\[
F(t_2, t_3, t_6, t_7 \rightarrow \emptyset) = ((\text{ready}. t_5. \text{running}. t_4)^* \cdot \text{ready}. (t_16 | (t_5. \text{running}. t_17)))
\]

We apply this evaluation for all possible combinations. When the function equals $\emptyset$ for a particular combination, it means that no correct state machine can be derived, thus this particular decision branch is not valid. When the evaluation is not $\emptyset$, the function returns a regular expression that represents the topology of the derived state machine. In such cases, one can easily transform the regular expression into a UML state machine – the transformation used is bijective.

In our case study, five behavioral derivations are possible (see figure 9).

\[\text{Fig. 9: Five valid state machines}\]

Let us note that such an approach enables to suppress isolated states. As no valid sequence of state and transitions lead to such states, they cannot appear in non-null regular expressions.

\[\text{Fig. 10: Eight possible structural derivations}\]

Recall that there is a one to one relation between the derived class and its state machine, a mismatch is clearly shown here: only five valid state machines are possible whereas eight classes can be obtained. This mismatch must come from a design failure in the structural variability.

If we take a closer look at the derived classes, we note that cases F, G and H feature either one of the specific alarm mode triggering operations, `startSoundAlarm()` or `startVisualAlarm()`, without the trigger `startAlarm()`. This contradicts the behavioral parts where all state machine feature

5.4. Derivation analysis

After calculating all possible derivations of the state machine, a comparison with the structural derivation is performed. The number of structural derivations may be greater than the number of possible behavioral derivations due to the topology of the state-machine. This construction provides an evaluation of the consistency of both structural and behavioral parts of the model with respect to the derivation process.

First, all derivations of the associated classes are calculated. To do that, we only need to calculate all possible combinations of variable elements that respect the constraints defined by variation groups. Because the number of structural variations is not great in a class (operation and property), this step is reasonably easy to perform. In our example, eight possible classes can be derived (see figure 10).
startSoundAlarm() and startVisualAlarm() only when both startAlarm() and stopAlarm() are present.

To solve this, one has to add a constraint in the system family model, to indicate that both startSoundAlarm() and startVisualAlarm() exist only if the alarm functionality is chosen (see the variation group on figure 11).

Fig. 11: Refined variation group specification

We may further analyze the result of the derivation process. We are left with five possible cases, A to E. Case E is the watch with no alarm, case D is the full watch (both sound and visual alarms), case C is the watch with visual alarm and case B is the watch with the sound alarm. There remains case A: it features a watch that has the generic alarm triggers startAlarm() and stopAlarm(), yet no specific alarm mode. This comes from an underspecification of our case study. In fact we assumed that whenever the alarm function is enabled, it would take either or both of the form sound or visual. Yet this was not fully accounted for in the VariationGroups. One should update the TriggerVariationGroup of figure 5 and change its kind to OneAmongSeveral to enforce that at least one specific alarm mode is chosen.

This shows how a carefully monitored derivation process can provide valuable information as to the consistency of the variability scheme introduced at both structural and behavioral levels. The example that we used in this paper, though very simple, shows that errors and underspecifications can easily appear when one has to deal with a variability-enabled model.

5.5 Overall derivation process

Figure 12 summarizes how the various mechanisms presented takes place into a monitored derivation process. The first step consists in calculating all possible regular expressions that may be engendered from the variable state machine. This step acts as a filter to eliminate those that do not respect the constraints expressed in the variation groups of the system family model (see 5.2 above). The second step consists in analyzing the coherence between structural and behavioral derivations. As shown in the case study, the analysis detects incoherencies (see 5.3). Based on this, one can more easily track down design errors or underspecifications and update the model. When the analysis sends no error message, we can be sure that the variability scheme is sound and that all possible derivations can be made.

Fig. 12: Monitoring of derivation

6. Conclusion

The approach presented in this paper belongs to a wide range of research works that address the problem of introducing variability within UML models and provide means of managing its complexity. We have found yet that few of these works deal with behavioral models. This is our main concern here, as we are developing design methods for real time and embedded systems for which behavioral modeling is crucial.

Our approach aims at providing as much tool support as possible to variability modeling. In a previous paper [17] we have described the overall process and how decision model could be constructed.
from the information structured in our VariationGroups. In this paper we described two additional mechanisms: 1) a propagation mechanism ensures that the impacts of variability is fully reflected system-wide based on a set of rules that can be tailored to user needs; 2) an evaluation mechanism enables to analyze what are the allowed derivation branches. Several aspects of these mechanisms can be enhanced: the propagation mechanism could take into account the semantics of the variability constraints instead of simply following model and meta model structural relationships. The evaluation of valid state machines should be extended to more complex state machines, which might reveal insufficiencies in the algebra and or method chosen. Finally the combinatorial explosion of calculating all possible structural derivations shall be dealt with - one may think of using constraint-solver here.

Our main goal here was to show that a carefully monitored derivation process provides valuable information to assess the consistency and completeness of a system family model, even for as simple cases as the one presented in this paper. Our approach is operational and supported with various plugins for the Papyrus open-source UML modeler [1]: an assistant helps the design of system family model and supports the propagation mechanism. Another tool provides information about the system family model: number of possible derivations, generation of the decision model. The last tool is an assistant to help the designer to derive a given product model from the system family model.

7. References
tur Congres, Amsterdam, 2000.