Understanding Decision-Oriented Variability Modelling

Deepak Dhungana Paul Grünbacher
Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University Linz, Austria
{dhungana | gruenbacher}@ase.jku.at

Abstract

Researchers and practitioners have been developing a wide range of techniques and tools to model and manage variability as a response to the heterogeneity of application areas and the diversity of implementation practices in different domains. In our own research we have been developing a tool-supported approach to decision-oriented variability modelling, which is highly customizable to domain-specific needs. In the past we have reported on our experiences on using the approach and its benefits in diverse industrial contexts. In this paper we present a more formal description of our approach and define the execution semantics of decision-oriented variability models.

1. Introduction

Variability is an emergent property of software systems and results from different design decisions taken to address requirements and contexts from different users. Experience from large-scale long-living systems shows that knowledge about variability is mostly tacit in nature and manifests itself in many different kinds of artefacts (documents, software components, test cases, configuration parameters, etc.) and different mechanisms supported by programming languages, architectural styles, design patterns, etc. Variability models have been proposed as a means of communication to deal with explicit documentation of tacit knowledge and better utilization of the flexibility and adaptability provided by a system.

The importance of variability in software systems and the necessity of making knowledge about variability explicit in models have already been identified as important research areas in software engineering. Depending on the background of different researchers, the needs of different industrial contexts and the kinds of systems under investigation, several variability modelling tools and techniques are already available [1, 5, 14, 15, 18, 19]. However, due to the broad spectrum of application areas and the diversity of implementation practices in different domains, a "standard approach" for dealing with variability will probably never exist. There are a lot of "island solutions" for variability modelling which either focus one particular level of abstraction or are monolithic and fixed to a certain grammar, with a set of predefined features. This hinders the widespread use of the existing approaches in different domains and application contexts. Despite the importance of variability modelling and the usage of such models in a wide array of contexts, researchers and practitioners are still struggling to find tools and techniques that best suit their modelling needs.

Feature modelling is probably the most prominent approach for modelling variability. Starting from FODA [13], the feature-oriented view of the world has already gone far beyond variability modelling and system documentation. Several formal interpretations (e.g. survey in [19]) of feature models and their applications have already been published. Today several variants of feature-based variability modelling tools and techniques are available.

A comparably smaller number of publications propose decision-oriented approaches to modelling variability. The idea of decision modelling in product lines is not new; it was introduced by Campbell et. al. [4, 2] in the early 1990s, where decisions were "*actions which can be taken by application engineers to resolve the variations for a work product of a system in the domain*" [2]. Forster et. al. [10], Schmid et. al. [18], Sellier et. al. [20] and others have been actively publishing their research results in this area. Astonishingly, researchers have not yet found a common basis on the notion of decisions. Some researchers follow decision modelling on a rather informal basis (e.g. using tables [18]), while others have already automated the decision making procedure by using executable descriptions and formal approaches.

We have incorporated a decision-oriented approach into our tool suite DOPLER [8] consisting of the tools DecisionKing [7, 6], ProjectKing [17] and ConfigurationWizard [16]. Here we describe decision models used in DOPLER tools more formally based on our experiences and feedback from industry. We provide a definition of the decision-oriented variability modelling language DoVML.

2. The basics of DoVML

Modelling the variability of software systems involves modelling the problem space (i.e., stakeholder needs or desired features) and the solution space (i.e., the architecture and the components of the technical solution). Separation of concerns based on problem space and solution space was also dealt with by Metzger et. al [14]. Our decision-oriented variability modelling language (DoVML) supports the modelling of the problem space using *decisions* and the solution space using *assets*.

The basic constructs for modelling variability using DoVML are depicted in figure 1. A Variability model is a set of *decisions*, *assets* and *rules*. Decisions can be organized in *groups*. The dependencies between decisions are expressed using *visibility conditions* and *validity conditions*. The dependencies among assets and decisions are established using *inclusion conditions*. *Visibility conditions*, *validity conditions*, and *inclusion conditions* are boolean expressions (the concrete syntax of the expression language can be defined by the modeller). Rules are comparable to constraints that ensure that certain conditions always hold.

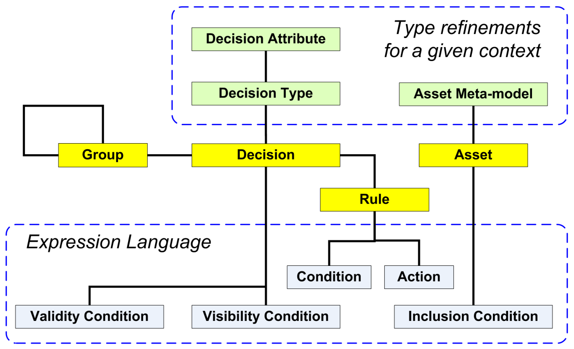


Figure 1. Constructs used in DoVML.

Product Line variability models built using DoVML are constructed such that they can be used for highly automated product derivation processes. The structure of the decision models and the concepts used therefor show high resemblance to process modelling approaches. As for example: *(i)* Visibility conditions are used to distinguish between decisions which are relevant for the user and the ones which are not. This guides the user through a product derivation process. *(ii)* Decision attributes like questions, descriptions and images are used to communicate decisions to the user. *(iii)* Rules are executed automatically to ensure the consistency of the decision making procedure.

2.1. The notion of a decision

A *decision* is a set of choices available at a certain point in time and arises whenever for a given goal there exist two or more ways of achieving it. Decisions can be used to represent the variation points in a product line model, and serve basically two purposes: *(i)* documenting and planning variability in the development phase and *(ii)* guiding users and automating product configuration during derivation phase. The process of taking a decision involves judging the merits of multiple options and selecting one of them for action (e.g., based on a consideration of customer requirements). In other terms a decision making process leads to the selection of a course of actions among several available alternatives.

Decisions are not independent of each other and cannot be made in isolation for two reasons: *(i)* Due to the dependencies surrounding a given decision, many decisions made earlier lead to new decisions and *(ii)* Many decisions are limited (constrained) depending on the context of already taken decisions.

In our modelling approach, we take care of two kinds of dependencies among decisions. Firstly, we need to be aware of the fact that not all decisions are equally important or relevant at a certain time. We therefore need constructs to model the hierarchy of decisions. Secondly, taking a certain decision may have implications on other decisions which also need to be considered (constraints). We therefore need to take care of the factors that influence the decision making process itself.

2.2. Decision vs. decision variable

For modelling purposes, we sometimes refer to a decision as a *decision variable*. A decision is a variable (like in programming languages) enriched with information regarding:

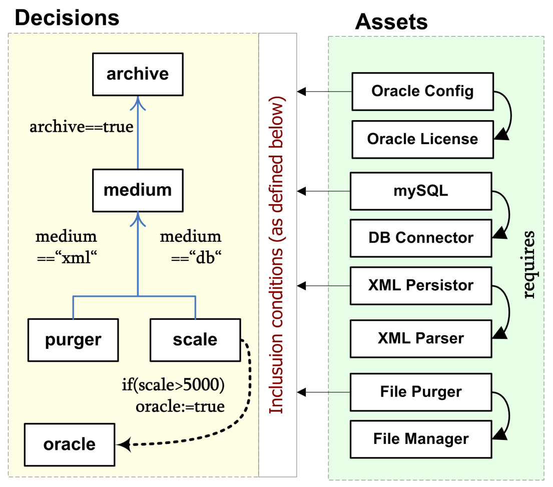
1. the set of possible values (including infinite sets, multiple ranges, and/or range constraints)

2. the specification of its position in the decision hierarchy (in relation to other available decisions)

3. the specification of the implications of taking the decision (on other decisions) and

4. labels and annotations (information for the user to better understand the decision).

Therefore, taking a decision is equivalent to binding a variable to a value.



Partial declaration of decisions (Problem Space)

Boolean decision **archive** visible if true;
Question: Do you want to archive daily records?

Enumeration decision **medium** visible if archive==true;
Question: Which medium should be used for archiving records?

Boolean decision **purger** visible if medium=="xml";
Question: Do you want old xml files to be automatically deleted?

Integer decision **scale** visible if medium=="db"; valid if scale>=0;
Question: Enter the anticipated number of daily records!
Post-script: if(scale>5000) oracle:=true;

Integer decision **oracle** visible if false;
// state variable, not visible to the user

Partial declaration of assets (Solution Space)

Component **XML Persistor** included if medium=="xml";
requires {XML Parser}

Component **mySQL** included if medium=="db" && scale<5000;
requires {DB Connector}

Component **File Purger** included if purger==true;
requires {File Manager}

Figure 2. Simplified representation of a variability model in DoVML.

2.3. The notion of an asset

Assets are used to describe the set of artefacts and their dependencies that are available in a certain development environment. The structure and organization of the solution space is specific to the domain/industrial context at hand, therefore the core of DoVML can be parameterized with an asset-meta model. Our approach doesn't assume fixed types of assets for modelling variability. By providing an abstract conceptual representation of structured data, the modeler defines the "modeling language" for the solution space. Due to lack of space in this paper, we omit the details about asset meta-modelling. In our modelling approach, the assets are linked to decisions via *inclusion conditions*, which are arbitrary boolean expressions built using the decision variables.

Figure 2 depicts a simplified representation of a variability model. It depicts the two key modelling elements (decisions and assets). The types of decisions in use (Boolean, Enumeration, Integer etc.) and the types of assets in use (Components, Resource, etc.) have been ommitted. For a better understanding of the terms and concepts presented in this paper, figure 2 is used as an running example. In the example, we assume a simple concrete syntax of the different expressions in use and the kind of relationships between different assets (e.g. requires) to be fixed.

2.4. Key concepts

In this paper the constructs of DoVML are defined using elementary set theory. We use the terms *types*, *variables* and *expressions* in the same way as in typed λ-calculus and functional programming languages. This means that expressions do not have side-effects and variables are bound to values. It also means that complex expressions are built from variables and simpler sub-expressions, by means of functions and operations. To give an abstract definition of decision models, it is not necessary to fix the concrete syntax in which the modeller writes the expressions, and thus we shall assume that such a syntax exists (together with well defined semantics). It is now possible in an unambiguous way to talk about the following:

1. The *type of a decision* υ is denoted by $\tau(\upsilon)$ and the *type of an expression* ε is denoted by $\tau(\varepsilon)$. For a type τ, we also use τ to denote the set of elements in τ.

2. The set of decisions involved in an expression ε is denoted by $\mathbb{V}(\varepsilon)$. This set of variables only includes the free variables, i.e., those which are not

bound internally in the expression (e.g. by local definition).

3. For a set of decision variables $\{\upsilon_1, \upsilon_2, .., \upsilon_n\}$, *the binding of the decisions* in the set is denoted by β=$\langle\upsilon_1$:η_1, υ_2:η_2, .., υ_n:$\eta_n\rangle$. It is required, that $\eta_i \in \tau(\upsilon_i)$.

4. Furthermore $\mathbb{E}_B(\mathtt{S})$ is defined as the set of Boolean expressions (terms and formulae), that can be built using the variables in the set S. In other words $\forall\varepsilon\in\mathbb{E}_B(\mathtt{S})$: $\tau(\varepsilon) \in \mathbb{B} \wedge \mathbb{V}(\varepsilon) \in \mathtt{S}$, where $\mathbb{B}$=$\{\mathtt{true}, \mathtt{false}\}$.

3. Variability modelling with DoVML

DoVML needs to be parameterized (configured) to the specifics of the domain, before it can be used to model the variability. Such configurability of the language provides us with the flexibility required to adapt the approach to the needs of different variability implementation practices. We therefore define Σ, $\mathbb{L}$, $\mathbb{A}$ and AMM as needed, where

Σ is a finite *set of data types*, specifying the types of variables to be used in the model, e.g. Boolean, Enumeration, String, Double, Character etc. This set can be extended with other types, as required by the domain. E.g., more complex compound data types (e.g. Date and Time) are also possible. In the overview depicted in figure 1, the set Σ is represented by *decision types*.

$\mathbb{L}$ is a finite *set of labelling functions* providing detailed information for every decision variable. Such annotations have no formal meaning, but are helpful in understanding the model. Examples of such labels are- description of υ, images and URLs to elaborate the meaning of υ to the user, the question which the user is asked etc. Use of labelling functions (as compared to unstructured text-tags) helps in better interpretation of the tags. In the overview depicted in figure 1, such labels represent the *decision attributes*.

$\mathbb{A}$ is a finite *set of actions*, which are carried out upon taking decisions defined in the decision model. Read only actions are used to validate the actual status of the decisions taken by the user (e.g. assertions that can be made in order to make sure that certain constraints are fulfilled). Other actions can be used to make changes in the model: variables can be bound to new values and other properties of decisions can be changed. The execution semantics of the action should be provided with its definition. Actions can be compared to domain-specific functions for manipulation of decisions.

AMM is the meta-model of the assets, whose variability needs to be modelled. It is comparable to “entity-relationship models” in relational databases. The asset meta-model specifies (defines) the language for describing the solution space.

A decision model (DM) is a set of decision variables of the defined types $\forall\upsilon \in \mathtt{DM} : \tau(\upsilon) \in \Sigma$. Every decision variable in a decision model is a unique identifier and can be bound to a certain set of values. The names have no formal meaning but they have huge practical importance for the readability of a decision model (just like the use of mnemonic names in traditional programming). The range of possible values is partly specified by the type of the variable.

Furthermore the decisions in DM are specified in more detail using f_{val}, f_{vis} and $\Re$ where

1. f_{val} is a validity function restricting the range of variables $\forall\upsilon \in \mathtt{DM} : f_{val}(\upsilon) \to \mathbb{E}_B(\mathtt{DM})$.

2. f_{vis} is a visibility function specifying the hierarchy of decisions $\forall\upsilon \in \mathtt{DM} : f_{vis}(\upsilon) \to \mathbb{E}_B(\mathtt{DM}\backslash\{\upsilon\})$.

3. $\Re$ is a set of rules in the form “if condition then action” (e.g. $\mathbb{E}_B(\mathtt{DM}) \Rightarrow \langle\upsilon_1 : \eta_1\rangle$, where a condition implies a binding).

Using the asset meta-model AMM defined for the domain at hand, we also create an asset model AM, which describes the set of available artefacts. We associate a function f_{inc} to every asset α in the asset model, which specifies when α needs to be included in the final product $\forall\alpha \in \mathtt{AM} : f_{inc}(\alpha) \to \mathbb{E}_B(\mathtt{DM})$.

3.1. Validity condition $f_{val}(\upsilon)$

The set of possible values of a variable specified by the type of the variable is often too broad. As an example let us consider a decision υ, where $\tau(\upsilon) = \mathbb{R}$. In order to further restrict the range of the variable, one can make use of a validity condition, a Boolean expression involving variables in DM, $f_{val}(\upsilon)\to\mathbb{E}_B(\mathtt{DM})$. A validity condition $f_{val}(\upsilon)$ of a decision can be seen as the post-condition which has to be fulfilled after υ is bound to a certain value. Using validity conditions, it is possible to specify multiple ranges too (e.g. $f_{val}(\upsilon)\to(\upsilon\geq\eta_1\wedge\upsilon\leq\eta_2)\vee(\upsilon\geq\eta_3\wedge\upsilon\leq\eta_4)$). It can therefore also be seen as a range constraint, which is evaluated before a variable binding can take place. A binding β=$\langle\upsilon$:$\eta\rangle$ is valid if $\eta\in\tau(\upsilon)\wedge f_{val}(\upsilon)$, for $\forall\upsilon\in\mathtt{DM}$.

Example: For a decision variable υ, $\tau(\upsilon)=\mathbb{R}$ the validity condition could be defined as $f_{val}(\upsilon)\rightarrow(\upsilon\texttt{\%2=0})$, which would mean, that only even numbers are valid values of the variable. In figure 2, we make use of a validity condition for decision `scale`, $f_{val}(\texttt{scale})\rightarrow(\texttt{scale}\geq 1)$, meaning that only positive number from $\mathbb{Z}$ are valid values of `scale`.

3.2. Visibility condition $f_{vis}(\upsilon)$

For each decision variable $\upsilon\in\texttt{DM}$, there exists a visibility function $f_{vis}(\upsilon)$, which specifies, when a certain decision can be taken by the user at a certain point in time during derivation. The visibility condition needs to be evaluated, before a value has been assigned to the variable υ, so the expression returned by $f_{vis}(\upsilon)$ must not contain the variable υ itself. In other words, $\forall\upsilon\in\texttt{DM}$: $f_{vis}(\upsilon)\in\mathbb{E}_B(\texttt{DM})\wedge\mathbb{V}(\varepsilon)\subseteq(\texttt{DM}\setminus\{\upsilon\})$.

Hierarchy based on visibility conditions: The hierarchy of decisions (the order in which the decisions need to be taken) is partly specified by f_{vis}. To elaborate on the effects of visibility conditions, we define a relationship $\diamond$ between decision variables with respect to their visibility conditions. A variable υ_1 is said to have a $\diamond$ relationship to another variable υ_2, if the variable υ_2 appears in the visibility condition of υ_1. This kind of relationship between variables, which is written as $\upsilon_1\diamond\upsilon_2$ (read as υ_1's visibility depends on υ_2) is given if $\upsilon_2\in\mathbb{V}(f_{vis}(\upsilon_1))$. $\diamond$ is non-reflexive (the visibility condition of a variable cannot depend on itself), strictly antisymmetric (variables cannot depend on each other) and transitive.

Example: In figure 2, the decisions are organized in a hierarchy based on their visibility conditions. Lets consider the decision regarding the medium to archive: $f_{vis}(\texttt{medium})\rightarrow\texttt{archive}$. The decision `medium` can be taken by the user only if the decision `archive` is bound to the value `true`. This implicitly requires, that the decision `archive` needs to be taken before `medium`. We can also note that $f_{vis}(\texttt{archive})\rightarrow\texttt{true}$ and $f_{vis}(\texttt{oracle})\rightarrow\texttt{false}$, which means these decisions are always/never visible to the user respectively.

3.3. State variables

Decisions which are never visible to the user, i.e. $f_{vis}(\upsilon)\rightarrow\texttt{false}$, are referred to as *state variables* and can be bound to their values only as a result of rules ($\Re$). Such decision variables can be used to keep track of different execution states of the model. They are bound to their values automatically as a result of executing the rules. Such rules help in aggregating values of decisions which have already been taken and allow to simplify complex expressions in models. For example (cf. figure 2) the decision variable `oracle` determining whether a oracle database is needed for the final system may be bound to a certain value automatically after the user decides on the size `(scale)` of the final system.

3.4. Specification of rules $\Re$

The effects of taking a decision (on other decisions) are modelled using a set of rules. Rules can be used basically for *(i) Assertion*, *(ii) Binding*, *(iii) Update* and *(iv) Information*. The semantic of rules used for assertion and binding is identical to constraints specified using boolean expressions in constraint satisfaction problems (CSPs). However, by using rules to update the model and to communicate to users at runtime, one can go beyond the borders of traditional constraints (as this is not the focus of constraints in CSPs). This also shows that variability models based on DoVML are created with the focus of an interactive product derivation process. The rules are specified in the form:

$$\texttt{if}\ \langle\texttt{condition}\rangle\ \texttt{then}\ \langle\texttt{action}\rangle,$$
where $\texttt{condition}\in\mathbb{E}_B(\texttt{DM})$ and $\texttt{action}\in\mathbb{A}$.

A rule is activated or triggered when its `condition` evaluates to `true`. Here we present a few examples of rules and their application. For the sake of simplicity in the examples, we assume the syntax of the rules to be similar to Pascal like programming languages. Rules could be used for:

(i) Assertion: Dependencies among decisions, where certain conditions always need to hold, e.g. a constraint in the form $(\upsilon_1=\eta_1)\Rightarrow(\upsilon_2=\eta_2)$ could be specified using the rule: $\texttt{if}(\upsilon_1=\eta_1)\texttt{then assert}(\upsilon_2=\eta_2)$ or simply $\texttt{assert}(\neg(\upsilon_1=\eta_1)\vee(\upsilon_2=\eta_2))$. The assert action is a read-only action. It does not change the value of the variables, but only makes sure that the condition holds.

(ii) Binding: Whenever there is a need to change the values of the variables we make use of binding actions. e.g. $\texttt{if}\ (\upsilon_1=\eta_1)\ \texttt{then setValue}(\upsilon_2,\eta_2)$. In general a binding action is comparable to a constraint as in CSPs, i.e. a condition implies a binding $\mathbb{E}_B(\texttt{DM})\Rightarrow(\beta=\langle\upsilon_1:\eta_1\rangle)$. In contrast to the assertion action, binding actions change the actual value of the decisions (i.e., they take decisions on behalf of the user). Here `setValue` is used as an example of a binding action (the actual syntax and semantics of all the actions is fixed when defining $\mathbb{A}$).

(iii) Update: Not only the values but also different attributes of decisions can be updated/manipulated using rules. As for example, depending on the

value of one decision, the validity condition of another decision might change, e.g. `if` $(\upsilon_1{=}\eta_1)$ `then update(`f_{val}`(`υ_2`)`$\rightarrow$`(`$\eta_2 \div 5 \neq 3$`))`. Such an update action can be used to change the specification of model at runtime. Modification of the decision model itself as an implication of the decisions taken by the user can however also lead to problems regarding the determinability of the decision making procedure.

(iv) Information: Rules can also be used for informative purposes. By defining actions like `inform`, or `display` one can also capture knowledge which is required for the user during product derivation. Such rules have no formal semantics, but can be very helpful to the user to improve guidance during derivation. Example usage scenarios for this would be the creation of recommender systems based on variability models e.g. `if` $(\upsilon_1{=}\eta_1)$ `then inform('It is recommended that ...')` [17].

3.5. Building an asset model

Asset models are instances of the asset-meta model describing the structure of the solution space. When building an asset meta-model the types of assets to be used, their attributes and dependencies among them can be defined. At this point, it is important to point out some peculiarities of the asset meta-models which we use in our approach.

Inclusion conditions: We associate a Boolean expression called inclusion condition to every asset α in the asset model `(AM)`. $\forall\alpha\in$`AM`: f_{inc}`(`α`)` $\rightarrow\mathbb{E}_B$(`DM`). Such an expression specifies the condition under which the asset α will be included in the final product. If an asset is always included in the system (e.g. utility classes, common libraries) then its inclusion condition is simply `true`. Considering the example presented in figure 2, the inclusion condition of the component `XMLPersistor` is defined as `medium==xml`. This means that the component `XMLPersistor` is included in the final product, if the decision `medium` is set to `xml`. The inclusion condition can be arbitrarily complex and can involve any number of variables, thus supporting not only 1:1 mappings between decisions and assets.

Basic dependency types: Often assets are not included or excluded from the final product directly because of decisions taken by the user but rather because of technical dependencies resulting from their implementation. For example (cf. figure 2), the component `FileManager` is included in the final product because it is required by the component `FilePurger`. In order to model such technical dependencies (functional and structural) we provide with a set of predefined relationship types (for automated interpretation of asset-dependencies). Examples of such basic relationship types are inclusion, exclusion, parent, child, predecessor, successor, implementation, abstraction etc. When specifying the asset meta-model for a certain organization, the modeler can define his own name for dependency types (e.g. "requires") and link it with a predefined type (e.g. "inclusion"). The naming of dependency types is similar to the concept of stereotypes in the UML.

When defining the semantics of decision-oriented variability models, we ignore $\mathbb{L}$, and f_{vis}, as they are primarily modeled with the focus of the product derivation process. We also don't care of the concrete syntax in which f_{val} and $\mathbb{A}$ are written. Further constructs like "roles of users", "configuration tasks" and project specific adaptations of the variability model [17] are out of scope of this paper.

4. Semantics of DoVML

A decision model represents the set of all possible `valid variable bindings` of the variables in the decision model, resolution of `DM` $\equiv$ Υ`(DM)`$=\{\beta_1,\beta_2,\ldots,\beta_n\}$, where n is possibly ∞ due to variables with infinite ranges. The process of taking decisions selects one possible binding from Υ. The resolution of a decision model is given by $\beta\in\Upsilon$.

A concrete binding $\beta\in\Upsilon$ can then be used for evaluating (calculating) the list of required assets. From the product derivation perspective, the assets can be seen as boolean variables, whose values are determined by the evaluation of their inclusion conditions. Every asset α can be interpreted as a boolean variable $\tau(\alpha)$=$\mathbb{B}$, whose binding is given by $\langle\alpha{:}f_{inc}(\alpha)\rangle$.

Furthermore, if asset dependencies are defined between assets α_1, α_2, α_3, such that α_1 `requires` α_2 `requires` α_3, then the dependency `requires` can be seen as a copy function, that assigns the same inclusion condition $f_{inc}(\alpha_1)$ to $f_{inc}(\alpha_2)$ and $f_{inc}(\alpha_3)$. i.e. α_1 `requires` α_2 `requires` $\alpha_3 \Rightarrow (f_{inc}(\alpha_3) \equiv f_{inc}(\alpha_2) \equiv f_{inc}(\alpha_1))$. The consequence is that if α_1 is included in the final product, then α_2 and α_3 are also included.

4.1. Interpreting/executing a variability model

The operational semantics of decision-oriented variability models can be explained by the algorithm in figure 3, which can interpret such variability models. The result of executing such a variability model is a set of taken decisions (binding of decision variables) and a set of assets required for the desired product.

Decision-making based on variability models (e.g.,

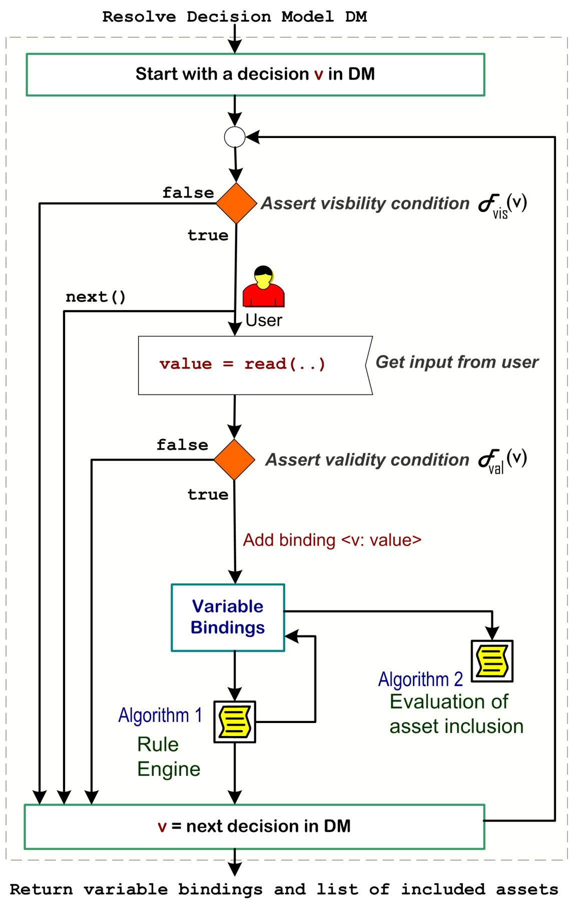

Resolve Decision Model DM
Start with a decision v in DM
false
Assert visbility condition $\sigma_{vis}(v)$
true
next()
User
value = read(..)
Get input from user
false
Assert validity condition $\sigma_{val}(v)$
true
Add binding <v: value>
Variable Bindings
Algorithm 2
Evaluation of asset inclusion
Algorithm 1
Rule Engine
v = next decision in DM
Return variable bindings and list of included assets

Figure 3. Overview of a sample algorithm for the execution of variability models.

as a part product derivation/configuration) is an interactive process. Decisions can either be visible or invisible to the user. The transition between these states is regulated by the evaluation of the visibility condition, which is triggered whenever a new variable binding takes place. All visible decisions are presented to the user. The variable binding takes place either as a result of user interaction or as a result of rules which are evaluated as required after a decision is taken. An asset can either be included in or excluded from the desired final product. The transition between these states occurs as a result of the evaluation of the inclusion condition of the assets.

Firstly, the visibility condition of each decision variable is evaluated. If the condition holds, then a question is presented to the user (possibly with other labels of the decision variable) so that the variable is better understood when taking the decision. The input from the user is evaluated against the validity condition. If the input was a valid one, then the variable is bound to the input value. Such a binding has two implications:

(i) *It triggers the rule engine*, which evaluates all the rules and executes them as necessary (overview of rule engine depicted in algorithm 1). Such rules can also cause a variable binding, which leads to a recursive call of the rule engine. So the execution of the action specified in the rule requires that the condition evaluates to `true`. The execution of the action can change the set of already bound variables; can however also only be informative. As the rule engine can trigger the evaluation of the rules again, it is important that there are no cyclic dependencies in the model. Cycles in the rules can be detected using standard cycle detection algorithms for graph like data structures.

(ii) *It triggers the evaluation of asset inclusion*, which is the process of figuring out which assets need to be included in the final product. The process (depicted in algorithm 2) consists of two phases: *(i)* evaluation of the inclusion condition and *(ii)* evalutation of asset dependencies. The set of included assets can then be used by domain-specific application generators simulators and deployment tools for further processing.

Algorithm 1 Sample evaluation of rules (rule engine)

```
Require: Binding $\beta \subseteq \{\upsilon_1:\eta_1, \upsilon_2:\eta_2, .., \upsilon_n:\eta_n\}$
  for all Rule $\rho$ in $\Re$ do
    if $\rho$.condition holds then
      if $\rho$.action is of type binding then
        $\beta = \beta \cup \{\langle \rho.\mathtt{action}.\upsilon : \rho.\mathtt{action}.\eta \rangle\}$
        re-evaluate all rules
      else
        domain-specific interpretation of $\rho$.action
      end if
    end if
  end for
  return Binding $\beta$
```

Algorithm 2 Sample evaluation included assets

```
Require: Binding $\beta \subseteq \{\upsilon_1:\eta_1, \upsilon_2:\eta_2, \ldots, \upsilon_n:\eta_n\}$
  initialize set of included assets L
  for all Asset $\alpha$ in AM do
    if $f_{inc}(\alpha)$ holds $\wedge$ then
      L = L $\cup$ $\{\alpha\}$
      {evaluate asset relationships}
      evaluate technical dependencies of $\alpha$
    end if
  end for
  return set of included assets L
```

5. Implementing DoVML

The approach described in this paper has been implemented in a meta-tool for modelling variability called DecisionKing [7, 9]. In order to reflect on the current implementation status of the tool (from the perspective of the modelling language features), let us consider the overview diagram depicted in figure 1. In DecisionKing we have realised the abstract core elements of DoVML (i.e. Decisions, Assets, Groups and Rules) by providing simple implementations for exemplification.

Types of decisions (data types): DecisionKing currently supports four basic types of decisions– `Boolean` decisions are used to simulate yes/no questions. `Number` decisions are used mostly for parameter values, where the user decides on a numerical value. These are comparable to the type "double" in programming languages. Other numerical types: integer, short etc can be simulated using number decisions. `String` decisions are used for similar purposes as number decisions. They correspond to the data type "String" in programming languages. `Enumeration` decision can be seen as arrays of strings. Such decisions are used whenever different alternatives to the same variation point need to be modeled.

Decision attributes (labelling functions): Currently we support three decision attributes to communicate the meaning of a decision to the user `Descriptions` are blocks of text (e.g., in HTML) is used to clarify the meaning of a decison. HTML also allows the easy integration of images, videos and animations to improve guidance of product derivation process. `Questions` are formulated in a concise way in the user's problem space language, such that the answer to that question implies the value of the decision. By making use of `Annotations` one can attach arbitrary information (in textual form) to a decision.

Expression language (functions and actions): We are currently using an expression language, which shows high syntactic resemblance to Pascal. One can make use of standard operators (e.g. $+, -, \div, *, =, \neq, \leq, \geq, <, >, \vee, \wedge$, etc.) to build expressions. DecisionKing provides an expression editor (with syntax highlighting and auto completion, cf. figure 4) to ease the modelling process. Apart from the standard operators we provide the following actions to query the value of decisions and build more complex expressions. `setValue(d1, p1)` is an assignment function, which assigns the value p1 to decision d1. `selectOption(ld1, op1)`, `deselectOption(ld1, op1)` are used to select/deselect an alternative in a enumeration decision. `contains(ld1, op1)` is a set operator which can be used in enumeration decisions to simulate $\subset, \subseteq, \in$ operations. `allow(ld1, op1)`, `disallow(ld1, op1)` are used to expand/restrict the set of possible values in a enumeration decision. `isTaken(d1)` is used to query, whether a decision has already been taken by the user. `reset()` is used to retract a taken decision. Retracting a decision also resets all its implications modelled in the rules. These functions and actions add syntactic sugar to the actual implementation of the engine that evaluates the rules. We are currently using a rule engine based on JBOSS Rules[1]. All rules written using the functions defined above are translated into their corresponding representation in drools[2] notation. As for example the following rule

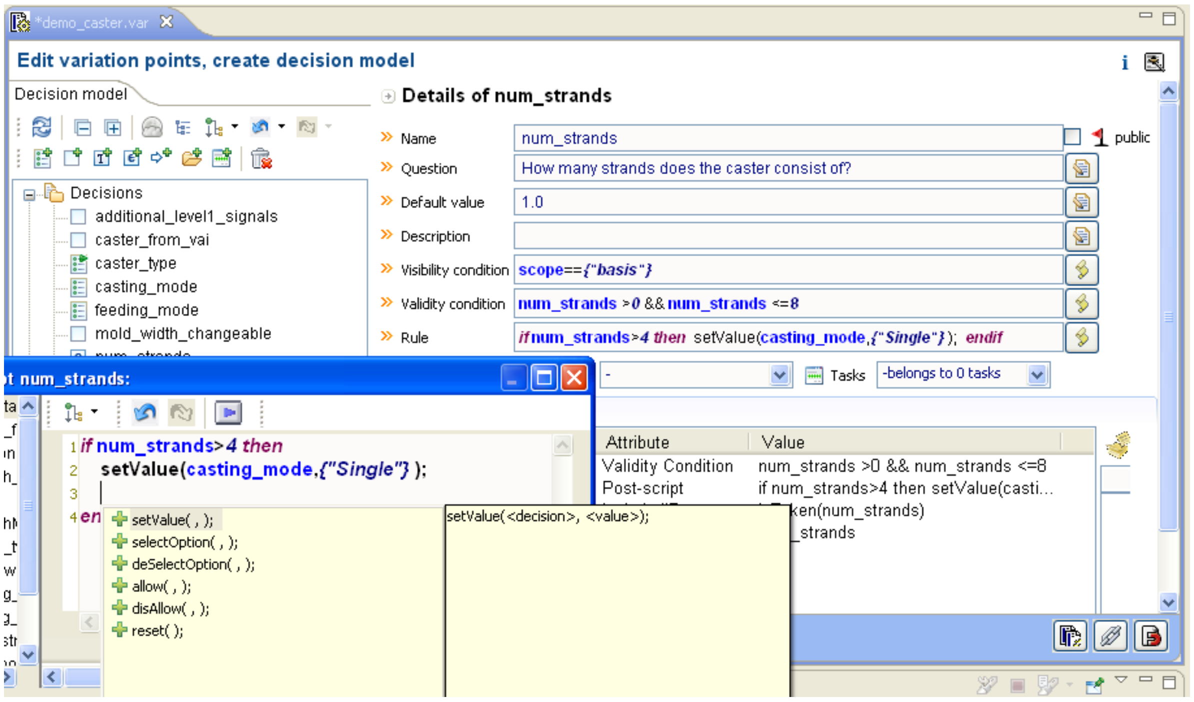
```
if (num_strands>4)then
  setValue(casting_mode, {"Single"});
endif
```

is automatically translated into drools rule:

```
rule "0"
salience 0
no-loop true
when
  num_strands:RuleNumberDecision(
    name == "num_strands",
    active==true)and
    eval(num_strands.getPValue()>4)then
      ArrayList<String> drools_a;
      i.identify();
      drools_a = new ArrayList<String>();
      drools_a.add("Single");
      i.set(0,"casting_mode",
        new ArrayList<String>(drools_a));
  end
```

Asset meta-models: As described earlier, for the description of the problem space, which is specific to implementation practices in different domains, our tool suite can be parameterized with a meta-model which is comparable to ER-models. Until now, we have created several such meta-models.
– *Siemens VAI*: For our industry partner we created a meta-model consisting of Components, Resources and Properties as asset types. Two dependency links (requires and contributes to) were used to describe the technical dependencies [7].
– *ERP System*: In order to model the variability of an enterprise resource planning (ERP) systems, we created a meta-model consisting of .NET "Plugins" as the basic asset type [21].
– *IAS System*: We also modeled the variability of an industrial automation system (IAS) to automate the runtime reconfiguration process [11].
– *DOPLER tool suite*: The variability of the DOPLER tool suite itself was modelled using DoVML. For this

[1] http://www.jboss.com/products/rules
[2] http://www.jboss.org/drools/



Edit variation points, create decision model

Decision model

Decisions
- additional_level1_signals
- caster_from_vai
- caster_type
- casting_mode
- feeding_mode
- mold_width_changeable

Details of num_strands

Name	num_strands
Question	How many strands does the caster consist of?
Default value	1.0
Description	
Visibility condition	scope=={"basis"}
Validity condition	num_strands >0 && num_strands <=8
Rule	if num_strands>4 then setValue(casting_mode,{"Single"}); endif

```
if num_strands>4 then
  setValue(casting_mode,{"Single"} );

en
```

```
setValue( , );
selectOption( , );
deSelectOption( , );
allow( , );
disAllow( , );
reset( );
```

```
setValue(<decision>, <value>);
```

Attribute	Value
Validity Condition	num_strands >0 && num_strands <=8
Post-script	if num_strands>4 then setValue(casti...

Figure 4. Modelling decision dependencies in DecisionKing.

purpose we create a meta-model consisting of Eclipse plugins, extension points and extension contributions as the asset types [12].

6. Summary and further work

In this paper we presented the details of our variability modelling approach based on decision modelling. Several publications in the past have already elaborated on the tools DecisionKing, ProjectKing and ConfigurationWizard, which are based on the modelling approach described in this paper.

Our modelling approach focuses on one of the primary goals of a product derivation process, i.e. the identification of the required assets to fulfil the needs of the customer specified in the form of taken decisions. This is however only one application area of variability models based on decisions. Flexibility and adaptability is introduced in our modelling approach by providing parameterization facilities for the language itself (e.g. by defining Σ, $\mathbb{L}$, $\mathbb{A}$ and $\mathtt{AMM}$ for each domain).

To illustrate the wide range of application areas, we have already used our approach and tools to automate the configuration of steel plant process automation software [7], to manage runtime adaptation of enterprise resource planning systems [21], to manage the lifecycle of industrial automation systems [11], and to monitor service-oriented systems at runtime [3].

We are currently working on more formal representations of decision-oriented variability models and their formal semantics. One longer term goal in this perspective is the formal definition and comparision to other available decision modelling approaches.

Apart from that, we are continuously extending the expression language used in our tool suite, which gives us the power to express variability constructs using functions at a higher level of abstraction. This includes implementation of different set operators, actions and other functions to model the dependencies among decisions/assets.

Ongoing work includes consistency checking and static analysis of decision-oriented models (e.g., by converting them into constraint satisfaction problems or petri nets) and further validation of the approach and tools in real world examples of our industry partner.

7. Acknowledgements

This work has been conducted in cooperation with Siemens VAI and has been supported by the Christian Doppler Forschungsgesellschaft, Austria. We would like to express our sincere gratitude to Stefan Wallner for his contribution in implementing the rule language based on JBOSS Rules.

References

[1] D. Batory. Feature models, grammars, and propositional formulas. In *9th International Software Product Line Conference (SPLC 2005)*, volume LNCS 3714, pages 7–20, Rennes, France, 2005.

[2] G. H. Campbell, S. R. Faulk, and D. M. Weiss. Introduction to synthesis. Technical report, Software Productivity Consortium, Herndon, VA, USA, 1990.

[3] R. Clotet, D. Dhungana, X. Franch, P. Grünbacher, L. López, J. Marco, and N. Seyff. Dealing with changes in service-oriented computing through integrated goal and variability modeling. In *Second International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2008)*, pages 43–52, Essen, Germany, 2008.

[4] S. P. Consortium. Synthesis guidebook. Technical report, SPC-91122-MC. Herndon, Virginia: Software Productivity Consortium, 1991.

[5] K. Czarnecki, S. Helson, and U. Eisenecker. Staged configuration using feature models. In R. Nord, editor, *Lecture Notes in Computer Science, Software Product Lines, Third International Conference (SPLC 2004)*, volume LNCS 3154, pages 266–283. Springer-Verlag, 2004.

[6] D. Dhungana, P. Grünbacher, and R. Rabiser. Decisionking: A flexible and extensible tool for integrated variability modeling. In *First International Workshop on Variability Modelling of Software-intensive Systems - Proceedings*, pages 119–128. Lero - Technical Report 2007-01, Limerick, Ireland, 2007.

[7] D. Dhungana, P. Grünbacher, and R. Rabiser. Domain-specific adaptations of product line variability modeling. In *IFIP WG 8.1 Working Conference on Situational Method Engineering: Fundamentals and Experiences*, Geneva, Switzerland, 2007.

[8] D. Dhungana, R. Rabiser, P. Grünbacher, K. Lehner, and C. Federspiel. Dopler: An adaptable tool suite for product line engineering. In *11th International Software Product Line Conference (SPLC 2007), Tool Demonstration*, volume Second Volume, pages 151–152, Kyoto, Japan, 2007. Kindai Kagaku Sha Co. Ltd.

[9] D. Dhungana, R. Rabiser, P. Grünbacher, and T. Neumayer. Integrated tool support for software product line engineering. In *Tool Demonstration, 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, Atlanta, Georgia, USA, 2007.

[10] T. Forster, D. Muthig, and D. Pech. Understanding decision models : Visualization and complexity reduction of software variability. In *Second International Workshop on Variability Modeling of Software-Intensive Systems*, volume 22, pages 111–119, 2008.

[11] R. Froschauer, D. Dhungana, and P. Gruenbacher. Managing the life-cycle of industrial automation systems with product line variability models. In *34th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, Parma, Italy, 2008.

[12] P. Grünbacher, R. Rabiser, and D. Dhungana. Product line tools are product lines too: Lessons learned from developing a tool suite. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, L'Aquila, Italy, 2008.

[13] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1990.

[14] A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *15th IEEE International Requirements Engineering Conference (RE'07)*, pages 243–253, New Delhi, India, 2007.

[15] V. Myllärniemi, M. Raatikainen, and T. Männistö. Kumbang tools. In *11th International Software Product Line Conference (SPLC 2007), Tool Demonstration*, volume Second Volume, pages 135–136, Kyoto, Japan, 2007. Kindai Kagaku Sha Co. Ltd.

[16] R. Rabiser and D. Dhungana. Integrated support for product configuration and requirements engineering in product derivation. In *33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'07)*, Lübeck, Germany, 2007.

[17] R. Rabiser, P. Grünbacher, and D. Dhungana. Supporting product derivation by adapting and augmenting variability models. In *11th International Software Product Line Conference (SPLC 2007)*, Kyoto, Japan, 2007.

[18] K. Schmid and I. John. A customizable approach to full-life cycle variability management. *Journal of the Science of Computer Programming, Special Issue on Variability Management*, 53(3):259–284, 2004.

[19] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature diagrams: A survey and a formal semantics. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 139–148, Minneapolis, MN, USA, 2006.

[20] D. Sellier and M. Mannion. Visualizing product line requirements selection decisions. In *11th International Software Product Line Conference (SPLC 2007), 1st International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2007)*, volume Second Volume, pages 109–118, Kyoto, Japan, 2007. Kindai Kagaku Sha Co. Ltd.

[21] R. Wolfinger, S. Reiter, D. Dhungana, P. Grünbacher, and H. Prähofer. Supporting runtime system adaptation through product line engineering and plug-in techniques. In *7th IEEE International Conference on Composition-Based Software Systems (ICCBSS)*, Madrid, Spain, February 2008. IEEE Computer Society.