Helsinki University of Technology Software Business and Engineering Institute
Research Reports 3

Teknillisen korkeakoulun ohjelmistoliiketoiminnan ja -tuotannon laboratorion
tutkimusraportti 3

Espoo 2007 HUT-SoberIT-A3

SOFTWARE AND SERVICES VARIABILITY MANAGEMENT WORKSHOP – CONCEPTS, MODELS AND TOOLS

Helsinki, Finland, April 2007

Proceedings

Tomi Männistö, Eila Niemelä and Mikko Raatikainen (eds.)

Helsinki University of Technology
Department of Computer Science and Engineering
Software Business and Engineering Institute

Teknillinen korkeakoulu
Tietotekniikan osasto
Ohjelmistoliiketoiminnan ja -tuotannon laboratorio

Distribution:
Helsinki University of Technology
Software Business and Engineering Institute
P.O. Box 9210, FIN-02015 TKK, Finland
Tel. +358-9-4511
Fax. +358-9-451 4958
E-mail: reports@soberit.hut.fi

ISBN 978-951-22-8747-5
ISSN 1458-6819

Multiprint Oy / Otamedia
Espoo 2007

Foreword

Software is becoming more pervasive and traditionally non-software products are increasingly turning into software-intensive products. Furthermore, the offering of many companies is moving towards service-oriented business instead of software or software-intensive products. At the same time, the amount of variability required is constantly increasing in order to improve customer satisfaction, cover larger and more heterogeneous markets, and so on. As more and more products and services must support variability, efficient variability management may become essential for surviving in competitive markets

Variability management is facing new challenges on its own and at least three trends can be identified. Firstly, binding of variability, i.e., the selection of the right variant is delayed later in the product's life cycle. Variability can also be re-bound, i.e., the product is reconfigured, and this can take place at runtime. Secondly, variability also concerns the quality attributes or non-functional properties of products. Product variants may even be differentiated solely in terms of their quality attributes. Thirdly, all variation possibilities cannot be assumed to be known beforehand, as, for example, new software or service components are developed by third parties and need to be incorporated into the product. Furthermore, products also need to adapt to the context of use. These trends are not separable but take place concurrently, which makes the preparation for them even more challenging.

In order to face the future challenges of variability management, clear conceptual foundation, rigorous modelling methods and languages, and different kinds of tools are needed to describe and manage the variability and to implement effective means for deriving products and services. The goal of this workshop is to explore and explicate the current status and ongoing work within the field of variability management by bringing together researchers and practitioners from different disciplines and application domains, and on this basis support fruitful transfer of knowledge.

The workshop received eleven submission of, which nine were accepted to the workshop. Five of the papers were accepted as full papers and four as short papers. The topics of the contributions focus on a set of common themes, such as modelling methods and tools and future directions in software variability management. Together the contributions form a basis for fruitful discussions on the emerging body of knowledge. We expect the workshop to make a relevant contribution in this respect by bringing together researchers with different backgrounds and linking the research efforts with industrial experiences and needs from different domains.

April, 2007

Tomi Männistö, Eila Niemelä and Mikko Raatikainen

Organizers

Workshop Chairs

Tomi Männistö, Helsinki University of Technology (TKK), Software Business and Engineering institute (SoberIT), Finland

Eila Niemelä, VTT Technical Research Centre of Finland, Finland

Mikko Raatikainen (local arrangement), Helsinki University of Technology (TKK), Software Business and Engineering institute (SoberIT), Finland

Program Committee

Jan Bosch, Nokia, Finland
Krzysztof Czarnecki, University of Waterloo, Canada
Alexander Felfernig, University of Klagenfurt, Austria
Albert Haag, SAP, Germany
Peter Knauber, Mannheim University of Applied Sciences, Germany
Kai Koskimies, Tampere University of Technology, Finland
Rene Krikhaar, ICT NoviQ, Netherlands
Thorsten Krebs, University of Hamburg, Germany
Charles Krueger, BigLever Software, USA.
Johan Lilius, Åbo Akademi University, Finland
John MacGregor, Robert Bosch GmbH, Germany
Mari Matinlassi, VTT, Finland
llkka Niemelä, Helsinki University of Technology, Finland
Klaus Pohl, Lero, Ireland, University of Duisburg-Essen, Germany
Christian Prehofer, Nokia, Finland
Pierre-Yves Schobbens, University of Namur, Belgium
Juha-Pekka Tolvanen, MetaCase, Finland
Rob van Ommering, Philips Research, Netherlands

Contents

Towards the Comparative Evaluation of Feature Diagram Languages

Patrick Heymans, Pierre-Yves Schobbens, Jean-Christophe Trigaux*, Raimundas Matulevičius, Andreas Classen and Yves Bontemps

PReCISE research centre, Computer Science Faculty, University of Namur
21, rue Grandgagnage – B-5000, Namur (Belgium)
{phe,pys,jtr,rma,aclassen,ybo}@info.fundp.ac.be

Abstract. This paper proposes a path to defragmenting research on feature diagram languages: (1) a global quality framework to serve as a language quality improvement roadmap; (2) a set of formally defined criteria to assess the semantics-related qualities of feature diagram languages; (3) a systematic method to formalise these languages and make them ready for comparison and efficient tool automation. The novelty of this paper resides in the latter point and the integration. The results obtained so far are summed up and future works are identified.

1 Introduction

During the last fifteen years or so, more than ten different *Feature Diagram* (FD) languages were proposed starting from the seminal work of Kang *et al.* on FODA back in 1990 [1]. An example of a FODA FD appears in Fig. 1. We assume the reader is familiar with the notation.

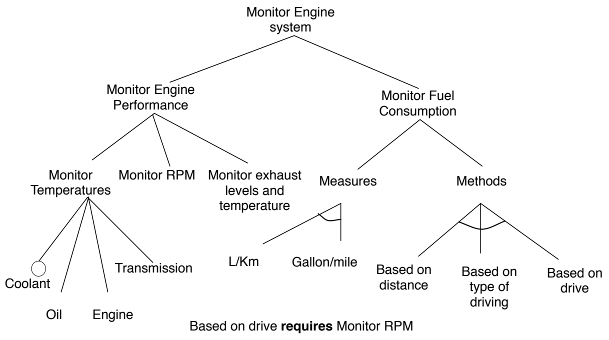


Fig. 1. FODA example (inspired from [2])

* Work supported by Walloon Region and FSE under FIRST Europe Objective 3 Project PLENTY EPH3310300R0462 / 215315

Since Kang *et al.*'s proposal, several extensions to FODA were devised [3–9] (see also Table 1). When looking at these FD languages, one immediately sees aesthetic differences (see, e.g., Fig.2). Although concrete syntax is an important issue in its own right [10], this work focusses on what is really behind the pictures: *semantics*. We noticed that proponents of FD languages often claimed for added value of their language in terms of precision, unambiguity or expressiveness. Nevertheless, our previous work [11–15] demonstrated that the terminology and evaluation criteria that they used to justify these claims were often vague, and sometimes even misleading. We also tried to give a precise meaning to the constructs of those languages.

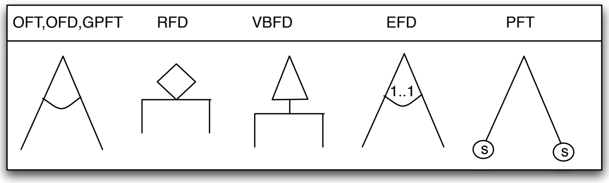


Fig. 2. Concrete syntaxes for *xor*-decomposition

Although we note that recent research has devoted more attention to the semantic foundations of these languages [16–22], we still lack concrete means to discriminate between these proposals.

This paper suggests a method to evaluate and compare FD languages focused on the study of their semantics. This method relies on formally defined criteria and terminology, based on the highest standards in formal language definition [23]. It is also situated with respect to SEQUAL [24, 25], a comprehensive framework for assessing and improving the quality of modelling languages.

In Section 2, we briefly present SEQUAL. Section 3 recalls good language definition principles from [23]. On these grounds, Section 4 continues with the definition of the criteria that our method aims to investigate: *expressiveness*, *embeddability* (also called *naturalness*), *succinctness* and (computational) *complexity*. The method is described in Section 5 and constitutes the main contribution of this paper. Section 6 summarises the results obtained so far [13–15]. The paper finishes by discussing the current limitations of the method and the remaining research challenges (Section 7), before it concludes (Section 8). This paper is short version of a technical report [26][1].

2 Quality of Models and Languages

Assessing and improving the quality of modelling is a complex and multidimensional task. A comprehensive view of the concerns involved is given in the SEQUAL (semiotic

[1] Available at `http://www.info.fundp.ac.be/~phe/docs/papers/TechRep_Eval_FP_of_FDL_06.pdf`

quality) framework, developed over the last decade by Krogstie *et al.* [25]. SEQUAL is based on a distinction between *semiotic levels*: syntactic, semantic and pragmatic. It adheres to a constructivistic world-view that recognises model creation as part of a dialog between participants whose knowledge changes as the process takes place.

SEQUAL is amenable to specific criteria and guidelines by tailoring. Its main advantages are that (1) it helps situate one's investigations within a comprehensive quality space, (2) it acts as a checklist of qualities to be pursued and (3) it recommends general guidelines on how to proceed.

Our investigation is targeted *semantic* and *pragmatic* qualities of FDs which we have found to be somehow neglected in the current state of the art. So doing, we will see that we inevitably interfere with the other qualities, mainly *syntactic* quality.

The problem we encounter is that representative objects of study – models – do not always exist, or at least are not easily available. And this is indeed the case for FDs which (1) are an emerging modelling paradigm, and (2) have the purpose of representing highly strategic company information. Since representative models[2] are almost nowhere to find, we concentrate on improving the quality of FD *languages*.

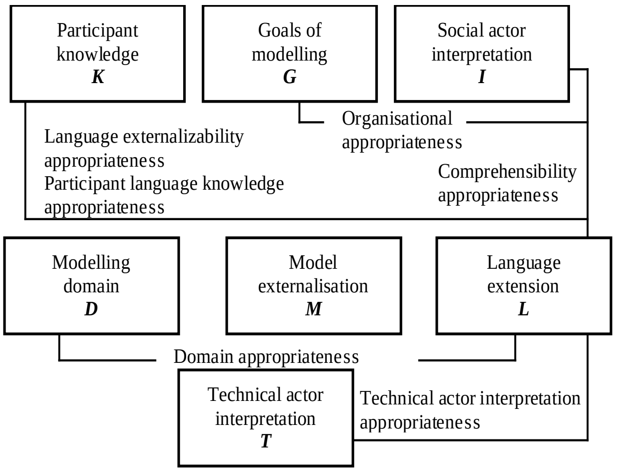

Participant knowledge ***K***
Goals of modelling ***G***
Social actor interpretation ***I***
Organisational appropriateness
Language externalizability appropriateness
Participant language knowledge appropriateness
Comprehensibility appropriateness
Modelling domain ***D***
Model externalisation ***M***
Language extension ***L***
Domain appropriateness
Technical actor interpretation ***T***
Technical actor interpretation appropriateness

Fig. 3. SEQUAL : Language Quality [24, 25]

SEQUAL has been adapted to evaluate language appropriateness [24] (see Fig. 3). Six quality areas were proposed. *Domain appropriateness* means that language *L* must be powerful enough to express anything in the domain *D*, and that, on the other hand it should not be possible to express things that are not in *D*. *Participant language knowledge appropriateness* measures how the statements of *L* used by the participants match the explicit knowledge *K* of the participants. *Knowledge externalisability appropriate-*

[2] Except illustrative examples used in research papers.

ness means that there are no statements in K that cannot be expressed in L. *Comprehensibility appropriateness* means that language users understand all possible statements of L. *Technical actor interpretation appropriateness* defines the degree to which the language lends itself to automatic reasoning and supports analysability and executability. Finally, *organisational appropriateness* relates L to standards and other needs within the organisational context of modelling.

Not being able to assess model qualities directly, our investigations were re-targeted at three main language qualities: *domain appropriateness*, *comprehensibility appropriateness* and *technical actor interpretation appropriateness*. The matching of the investigated criteria wrt these qualities is further discussed in Section 7. In the next section, we will first introduce the basic notions behind these criteria (Section 3), and then the criteria themselves (Section 4).

3 Formal definition of visual languages

In [23], Harel and Rumpe recognise that: *"Much confusion surrounds the proper definition of complex modelling languages [...]. At the root of the problem is insufficient regard for the crucial distinction between syntax and true semantics and a failure to adhere to the nature and the purpose of each."* [23] Although they are far less complex than, e.g., the UML[3], we demonstrated in previous papers [11–14] that FDs were also "victims" of similar "mistreatments".

Harel and Rumpe make it clear that the unambiguous definition of any modelling language must consist of three equally necessary elements: a *syntactic domain* ($\mathcal{L}$), a *semantic domain* ($\mathcal{S}$) and a *semantic function* ($\mathcal{M}$) (see Fig. 4). All three should be defined through explicit, rigid and unambiguous rules, hence the use of mathematics.

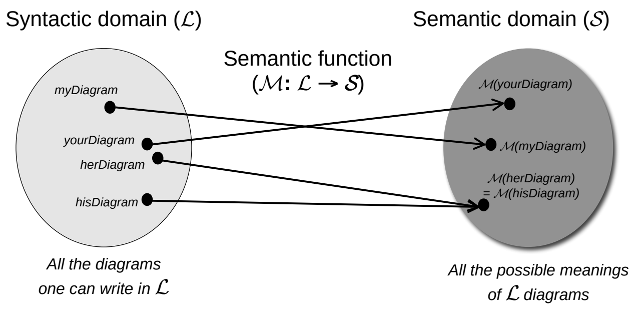

Syntactic domain ($\mathcal{L}$)
Semantic domain ($\mathcal{S}$)
Semantic function
($\mathcal{M}: \mathcal{L} \to \mathcal{S}$)
myDiagram
yourDiagram
herDiagram
hisDiagram
$\mathcal{M}$(yourDiagram)
$\mathcal{M}$(myDiagram)
$\mathcal{M}$(herDiagram) = $\mathcal{M}$(hisDiagram)
All the diagrams one can write in $\mathcal{L}$
All the possible meanings of $\mathcal{L}$ diagrams

Fig. 4. The 3 constituents of a formal language

During our survey, we could observe that many FD languages were never formally defined. Maybe, some answers to why this is so are given in [23] where the authors point

[3] In [23], one of Harel and Rumpe's main motivations is to suggest how to improve the UML.

out of set of frequent misconceptions about formal semantics, e.g., “Semantics is the metamodel”, “Semantics is dealing with behaviour”, “Semantics is being executable”, “Semantics means looking mathematical”, etc. This folklore is demystified [23]. For now, we turn to the definitions of $\mathcal{L}$, $\mathcal{S}$ and $\mathcal{M}$.

3.1 Syntax

Concrete syntax is the *physical representation* of the data (on screen, or on paper) in terms of lines, arrows, closed curves, boxes and composition mechanisms involving connectivity, partitioning and “insideness” [23].

Although discouraged by best pratice, most of the (informal) definitions of the semantics of FDs we found in the literature were based on concrete syntax, usually discussed on FD examples. Most of the time, a substantial part of the semantics was implicit, leaving it to the diagrams to “speak for themselves”.

The *abstract syntax* ($\mathcal{L}$) is a representation of data that is independent of its physical representation and of the machine-internal structures and encodings. It thus makes the syntactic rules simpler and more portable. The set of all data that comply with a given abstract syntax is called the *syntactic domain*.

In [13, 14], we provided an abstract syntax (and semantics) for several FD languages at once through a generic mathematical structure we called FFD (see Fig. 5 and Table 1). $\mathcal{L}_{FFD}$ has 4 parameters reflecting the 4 abstract syntax variation points we observed among languages: the *graph type* ($GT = \mathsf{TREE}$ or DAG[4]), the *node types* (NT, i.e. what decomposition operators can be used: *and*, *xor*, *or*,...), the additional *graphical constraint types* used (GCT, usually `requires`/$\Rightarrow$ and `mutex`/$|$), and the *texual constraint language* (TCL, usually Composition Rules (CR) [1]).

3.2 Semantics

The *semantic domain* ($\mathcal{S}$) *“[...] specifies the very concepts that exist in the universe of discourse. As such, it serves as an abstraction of reality, capturing decisions about the kinds of things the language should express”.* $\mathcal{S}$ is a mathematical domain built to have the same structure as the real-world objects the language is used to account for, up to some level of fidelity. The semantic domain that we have proposed for FODA-inspired languages is named *PL* (Product Lines) [13, 14]. It is recalled in Definition 1. It assumes that FDs are graphs whose nodes (N) represent features and where P, a subset of N, is the set of features that the user considers relevant. We call P the set of *primitive features/nodes*[5]:

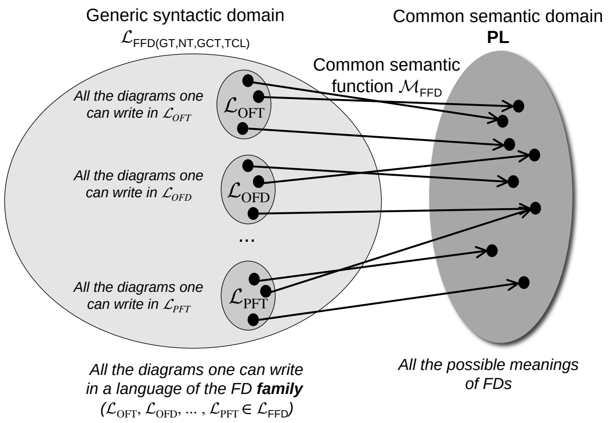
Definition 1 (Configuration, Product, Product Line). *(1) A* configuration *is a set of nodes, i.e., any element of $\mathcal{P}N$. (2) A* product *is a configuration that contains only primitive features, i.e., any element of $\mathcal{P}P$. (3) A* product line *is a set of products, i.e., any element of $PL = \mathcal{P}\mathcal{P}P$.*

[4] Directed Acyclic Graph.

[5] Hence, *primitive* nodes and *leaf* nodes are different concepts, although the former usually includes the latter, but can include intermediate nodes as well; this is up to the modeller.

Other formalisations [16–22] chose semantic domains different from *PL*, for example using lists instead of sets [22] or keeping the full shape of the FD [19]. How to compare *PL* with other semantic domains will be discussed in Section 5.

The *semantic function* $\mathcal{M} : \mathcal{L} \rightarrow \mathcal{S}$ eventually assigns a meaning in $\mathcal{S}$ to each syntactically correct diagram d, noted $\mathcal{M}[\![d]\!]$. Again, a mathematical definition is recommended. In [13, 14], we defined a generic semantic function ($\mathcal{M}_{FFD}$) giving a semantics to several FD languages at once (see Fig. 5).



Generic syntactic domain
$\mathcal{L}_{\text{FFD(GT,NT,GCT,TCL)}}$
Common semantic domain
PL
Common semantic function $\mathcal{M}_{\text{FFD}}$
All the diagrams one can write in $\mathcal{L}_{OFT}$
$\mathcal{L}_{\text{OFT}}$
All the diagrams one can write in $\mathcal{L}_{OFD}$
$\mathcal{L}_{\text{OFD}}$
...
All the diagrams one can write in $\mathcal{L}_{PFT}$
$\mathcal{L}_{\text{PFT}}$
All the diagrams one can write in a language of the FD **family**
($\mathcal{L}_{\text{OFT}}, \mathcal{L}_{\text{OFD}}, \ldots, \mathcal{L}_{\text{PFT}} \in \mathcal{L}_{\text{FFD}}$)
All the possible meanings of FDs

Fig. 5. Semantics for a family of FD languages

Since $\mathcal{M}$ is a *function*, there is at most one semantics for each diagram. Ambiguity in this context is therefore not possible. The term “ambiguity” was not always properly used in the surveyed literature. For example, FODA FDs have been criticised for being ambiguous [8]. However, having reconstructed a proper formal semantics from the original plain English definition [1], we could check that this was not the case [11].

Finally, the semantic function should be *total*, that is, it should not be possible to have a diagram in $\mathcal{L}$ which is not given a meaning in $\mathcal{S}$ by $\mathcal{M}$. The converse question (is every element in $\mathcal{S}$ expressible by a diagram in $\mathcal{L}$?) is called the *expressiveness* of a language and is another term used confusingly in the literature. It is clarified, together with other comparison criteria, in the next section.

4 Comparison criteria

When a language receives a formal semantics, it can then be evaluated according to various objective criteria. We first address (computational) *complexity*. In a formal language, we can precisely define *decision problems*, i.e., tasks to be automated. A mathematical definition of the tasks is necessary to prove the correctness of algorithms. It also allows to study complexity, thereby assessing their scalability. Results give an indication about the worst case, and how to handle it. Heuristics taking into account the most usual cases can be added to the backbone algorithm, to obtain practical efficiency.

In [13], we studied the complexity of a selection of FD-related decision problems: (1) *satisfiability*: given a diagram d, is $\mathcal{M}[\![d]\!] = \emptyset$ true? (2) *equivalence*: given two diagrams d_1 and d_2, is $\mathcal{M}[\![d_1]\!] = \mathcal{M}[\![d_2]\!]$ true? (3) *model-checking* (called *product-checking* for FDs): given a product c and a diagram d, is $c \in \mathcal{M}[\![d]\!]$ true? (4) *intersection*: compute a new diagram d_3 such that $\mathcal{M}[\![d_3]\!] = \mathcal{M}[\![d_1]\!] \bigcap \mathcal{M}[\![d_2]\!]$. (5) *union*: compute a new diagram d_3 such that $\mathcal{M}[\![d_3]\!] = \mathcal{M}[\![d_1]\!] \bigcup \mathcal{M}[\![d_2]\!]$. (6) *reduced product*: compute a new diagram d_3 such that $\mathcal{M}[\![d_3]\!] = \{c_1 \cup c_2 | c_1 \in \mathcal{M}[\![d_1]\!], c_2 \in \mathcal{M}[\![d_2]\!]\}$.

When languages, in addition to having a formal semantics, also share a *common* semantic domain ($\mathcal{S}$), we can compare them with additional criteria. We use three common criteria:

- *expressiveness*: what can the language express?
- *embeddability* (or *macro-eliminability*): when translating a diagram to another language, can we keep its structure?
- *succinctness*: how big are the expressions of a same semantic object?

Formal semantics opens the way for a fully formal definition and objective assessment of these criteria. For example, Def. 2 naturally formalizes *expressiveness* as the part of a languages's semantic domain that its syntax can express. Fig. 6 illustrates it.

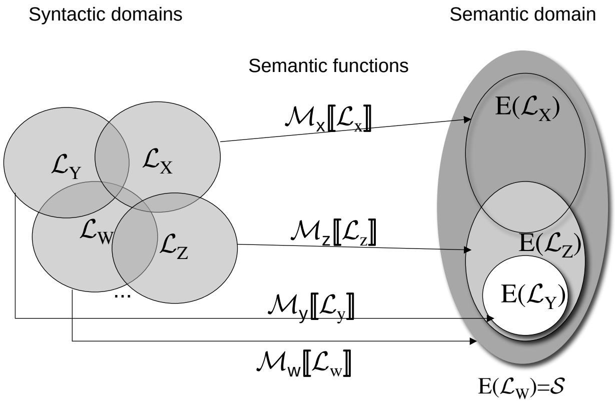
Definition 2 (Expressiveness).
The expressiveness *of a language $\mathcal{L}$ is the set $E(\mathcal{L}) = \{\mathcal{M}[\![d]\!] | d \in \mathcal{L}\}$, also noted $\mathcal{M}[\![\mathcal{L}]\!]$. A language $\mathcal{L}_1$ is* more expressive *than a language $\mathcal{L}_2$ if $E(\mathcal{L}_1) \supset E(\mathcal{L}_2)$. A language $\mathcal{L}$ with semantic domain $\mathcal{S}$ is* expressively complete *if $E(\mathcal{L}) = \mathcal{S}$.*

Since languages compete for expressiveness, it often happens that they reach the same maximal expressiveness (like $\mathcal{L}_W$ in Fig. 6). This is for instance the case for programming languages, that are almost all Turing-complete and can thus express the same computable functions. Consequently, we need finer criteria than expressiveness to compare these languages.

In [13], we recalled equally formal definitions of embeddability and succinctness, which are widely accepted criteria in the formal methods community. We cannot reproduce them here for lack of space, so we just present them informally and motivate them. The results obtained by applying them to FDs are detailed in Section 6.

Embeddability is of practical relevance because it questions the existence of a transformation from one language to the other which preserves the whole shape of the diagrams and generates only a linear increase in size. This way, traceability between the two diagrams is greatly facilitated and tool interoperability is made more transparent. Furthermore, limiting the size of diagrams helps avoiding tractability issues for reasoning algorithms taking the diagrams as an input. Most importantly, embeddability can also exist between a language and a subset of itself. A language that is non-trivially self-embeddable [13] is called *harmfully redundant*. This means that it is unnecessarily complex: all diagrams can be expressed in the simpler sublanguage without loss of structure and with only a linear increase in size.

In case linear translations are not possible, the blow-up in the size of the diagram must be measured by *succinctness*. If $\mathcal{L}_1$ is more succinct than $\mathcal{L}_2$, this usually entails that $\mathcal{L}_1$'s diagrams are likely to be more readable. Also, if one needs to translate from $\mathcal{L}_1$ to $\mathcal{L}_2$[6], succinctness will be an indicator of the difficulty to maintain traceability between the orginal and the generated diagram. Traceability of linear translations is usually easier but is likely to become more difficult as the size of the generated diagrams grows bigger. However, this should not be concluded too hastily since succinctness does not provide information on the structure of the generated diagrams[7]. In this sense, succinctness is a coarser-grained criteria than embeddability.



Syntactic domains
Semantic domain
Semantic functions
$\mathcal{L}_Y$
$\mathcal{L}_X$
$\mathcal{L}_W$
$\mathcal{L}_Z$
...
$\mathcal{M}_x[\![\mathcal{L}_x]\!]$
$\mathcal{M}_z[\![\mathcal{L}_z]\!]$
$\mathcal{M}_y[\![\mathcal{L}_y]\!]$
$\mathcal{M}_w[\![\mathcal{L}_w]\!]$
$E(\mathcal{L}_X)$
$E(\mathcal{L}_Z)$
$E(\mathcal{L}_Y)$
$E(\mathcal{L}_W)=\mathcal{S}$

Fig. 6. Comparing expressiveness

5 A Comparison Method for FD languages

In order to compare FD languages $X_1, \ldots, X_n$ according to the criteria exposed in the previous section, we need formally defined languages. That is, for language X_i, we need $\mathcal{L}_{X_i}$, $\mathcal{S}_{X_i}$ and $\mathcal{M}_{X_i}$. To compare expressiveness, embeddability and succinctness, we also need to have $\mathcal{S}_{X_1} = \mathcal{S}_{X_2} = \ldots = \mathcal{S}_{X_n}$. Unfortunately, this ideal situation almost never occurs in practice. Instead, we have to cope with:

- languages that have *no formal semantics* (this is the most frequent case [13, 14]),
- languages with a *formal semantics defined in a different way from* [23],
- or languages with a *formal semantics* compliant with [23] *but different semantic domains*.

[6] E.g., because a tool for achieving some desired functionality is only available in $\mathcal{L}_2$.
[7] However, looking at the transformation's definition will provide the information.

Hence, the overall comparison process should be carried out in two steps: (1) make the languages suitable for comparison, (2) make the comparisons. We now detail the first step.

Let X_1 be the language we want to compare with the others $(X_2, ..., X_n)$ which, we assume, are fully and clearly formalised according to [23] and have identical semantic domains. We distinguish three cases:

5.1 Case 1: X_1 has no formal semantics

There are two alternatives:

- The first alternative is to define the syntax and semantics for each FD language individually following [23]. That is, we define X_1 independently from $X_2, ..., X_n$. This is what we did in [11] where we formalised FODA FDs (OFT) [1]. FORM FDs (OFD) [3] are treated the same way in [26].
- The second alternative is to make scale economies and define several languages at once. In [14], we observed that most of the FD languages largely share the same goals, the same constructs and, as we understood from the informal definitions, the same (FODA-inspired) semantics. For this reason, we proposed to define not one FD language but a family of related FD languages (see Fig. 5). We defined a parametric abstract syntax, called FFD, in which parameters correspond to variations in $\mathcal{L}_{X_1}, ..., \mathcal{L}_{X_n}$. This definition follows, but slightly adapts, the principles of Section 3. The semantic domain (PL) and semantic function are common to all FD variants, maximizing semantic reusability. With this method, we are confined to handle languages whose only significant variations are in abstract syntax. For languages with very different semantic choices, e.g. [19], it is much harder to describe (and justify) the introduction of variation points in the semantics. Then, we should rather follow either the first alternative in Case 1 if the language is informal, or Cases 2 or 3 otherwise.

5.2 Case 2: X_1 has formal semantics but $\mathcal{L}_{X_1}$, $\mathcal{S}_{X_1}$ and $\mathcal{M}_{X_1}$ need to be clarified

Another frequent case is when X_1 actually has a formal semantics, but irrespective of [23]. That is, we cannot see explicit and self-contained mathematical definitions of $\mathcal{L}_{X_1}$, $\mathcal{S}_{X_1}$ and $\mathcal{M}_{X_1}$. Typically, $\mathcal{L}_{X_1}$ is clear and self-contained, but $\mathcal{S}_{X_1}$ and $\mathcal{M}_{X_1}$ are not. Most of the time, the semantics of X_1 is given by describing a transformation of X_1's diagrams to another language, say W, which is formal. W does not even need to be a FD language, and usually it is not. Therefore, the semantic domain might be very different from the one intuitively thought of for FDs. The main motivation for formalising this way is usually because W is supported by tools. The problem is that these kinds of "indirect", or tool-based, semantics complicate the assessment of the language[8].

Several proposals of this kind for FDs can be found in recent work [17–22]. We thus need to reformulate the semantics of those languages. In [15], we treated the FD language proposed by van Deursen and Klint [22] (renamed vDFD) before comparing

[8] Even more if W's semantics also does not follow [23].

it to FFD. The main difference w.r.t. Case 1 is that here formalisation decisions are usually much more straightforward since they have already been made. However, they might be hard to dig out if they are coded in some tool. Also, formalisations are not necessarily error-free, and errors can thus be discovered when re-formalising [15].

5.3 Case 3: X_1 has formal semantics with clear $\mathcal{L}_{X_1}$, $\mathcal{S}_{X_1}$ and $\mathcal{M}_{X_1}$ but $\mathcal{S}_{X_1} \neq \mathcal{S}_{X_2}, ..., \mathcal{S}_{X_n}$

The third and last case is when we have a clear and self-contained mathematical definition of $\mathcal{L}$, $\mathcal{S}$ and $\mathcal{M}$ for all languages (either from the origin, or having previously gone through Case 1 or 2) but the semantic domains of the languages differ. We thus need to define a relation between the semantic domains. We met this problem, for instance, when comparing vDFD with FFD [15]. On the one hand, we had $\mathcal{S}_{FFD} = PL = \mathcal{PP}P$ (sets of sets of nodes), and on the other, $\mathcal{S}_{vDFD} = OON$ (lists of lists of nodes). The latter introduces an order relation on features, and one on products. Comparing languages with different semantic domains is actually possible, but it requires preliminary work which is now explained.

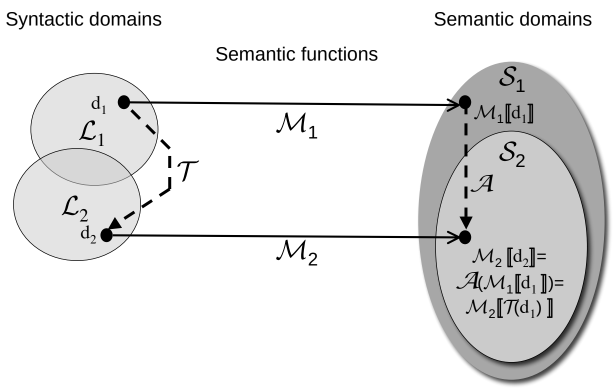


Fig. 7. Abstracting a semantic domain

We need to define an *abstraction function* ($\mathcal{A}$ in Fig. 7) whose purpose is to remove extra information from the richer domains and keep the "core" of the semantic domain, where we will perform the comparisons. We used such a function to remove the ordering of features and products from $\mathcal{S}_{vDFD}$ [15]. However, the question of the relevance of this discarded information remains and should be studied carefully. A fairly general case is illustrated in Fig. 7, where domain $\mathcal{S}_1$ contains more information than $\mathcal{S}_2$; we then take $\mathcal{S}_2$ as the common domain. $\mathcal{A}$ removes extra information from elements of $\mathcal{S}_1$ and maps them in $\mathcal{S}_2$. It then makes sense to look for quasi-translations $\mathcal{T} : \mathcal{L}_1 \rightarrow \mathcal{L}_2$ between their syntactic domains. They are translations for the abstracted semantics $\mathcal{A} \circ \mathcal{M}_1$, and can thus be used to compare languages for expressiveness, embeddability or succinctness. Hence, if we apply $\mathcal{T}$ to a diagram d_1 in the syntactic domain $\mathcal{L}_1$ we will obtain a diagram d_2 in the syntactic domain $\mathcal{L}_2$ with the same abstracted semantics. Semantically, if we apply the semantic function $\mathcal{M}_1$ to d_1 and then the abstraction function $\mathcal{A}$, we will map on the same element of $\mathcal{S}_2$ as if we apply $\mathcal{T}$ to d_1 and then $\mathcal{M}_2$: $\mathcal{A}(\mathcal{M}_1[\![d_1]\!]) = \mathcal{M}_2[\![\mathcal{T}(d_1)]\!]$.

When applied to more than two languages, this method will create many semantic domains related by abstraction functions. The abstraction functions can be composed and will describe a category of the semantic domains. At the syntactic level, the translations can also be composed to yield expressiveness and succinctness results. Similarly, the composition of embeddings yields an embedding.

6 Language Evaluation Results

We summarise the results obtained by applying our general comparative semantics method. For the languages defined generically with FFD (see Table 1), the details and proofs can be found in [13]. The treatment of vDFD [22] is found in [15].

Survey short name	**GT**	**NT**	**GCT**	**TCL**
OFT [1]	TREE	$and \cup xor \cup \{opt_1\}$	$\emptyset$	CR
OFD [3]	DAG	$and \cup xor \cup \{opt_1\}$	$\emptyset$	CR
RFD [4]=VBFD [9]	DAG	$and \cup xor \cup or \cup \{opt_1\}$	$\{\Rightarrow, \mid\}$	CR
EFD [7,8]	DAG	$card \cup \{opt_1\}$	$\{\Rightarrow, \mid\}$	CR
GPFT [5]	TREE	$and \cup xor \cup or \cup \{opt_1\}$	$\emptyset$	CR
PFT [6]	TREE	$and \cup xor \cup or \cup \{opt_1\}$	$\{\Rightarrow, \mid\}$	$\emptyset$
VFD [13]	DAG	$card$	$\emptyset$	$\emptyset$

Table 1. FD languages defined through FFD

6.1 Complexity

For FDs, solving all the standard problems of Section 4 turns out to be practically useful:

- *Equivalence* of two FD is needed whenever we want to compare two versions of a product line (for instance, after a refactoring). When they are not equivalent, the algorithm can produce a product showing their difference. For FD languages based on DAGs, and that allow non-primitive features, such as OFD, EFD, VFD, this problem is Π_1-complete [13] (just above NP-complete).
- *Satisfiability* is a fundamental property. It must be checked for the product line but also for the intermediate FDs produced during a staged configuration [20]. For FD languages based on DAGs, this problem is NP-complete.

– *Model-checking* verifies whether a given product (made of primitive features) is in the product line of a FD. It is not as trivial as expected, because the selection performed for non-primitive nodes must be reconstructed. This gives an NP-complete problem. When recording this selection, the problem becomes linear again.
– *Union* is useful when parallel teams try to detect feature interference in FDs. Their work can be recorded in separate FDs, the union of which will represent the validated products. For FD languages based on DAGs, this problem is solved in linear time, but the resulting FD should probably be simplified for readability. *Intersection* and *reduced product* are similar.

The complexity results show the role of non-primitive features. On one hand, it is useful to record them to accelerate the checking of products. However, they should not become part of the semantics since this would restrict the expressiveness and strongly reduce the possible transformations of diagrams.

6.2 Expressiveness

The distinction between languages that only admit trees and the ones that allow sharing of features by more than one parent (DAGs or vDFD) turns out to be important. While tree-shaped languages are usually incomplete, OFD [3] are already expressively complete without the constraints, and thus *a fortiori* RFD [4], EFD [7, 8] and VFD [13]. vDFD are “almost” trees in that only terminal features (i.e. the leaves) can have multiple parents (justifications), but this is sufficient to obtain expressive completeness.

In contrast, tree-shaped diagrams turned out to be expressively incomplete; in particular, OFT [1] cannot express disjunction. This justifies *a posteriori* the proposal [9] (VBFD) to add the *or* operator to OFT. But even so, we do not attain expressive completeness: this language is still unable to express $card_3[2..2]$, the choice of two features among three[9]. This justifies similarly the proposal [7] (EFD) to use the *card* operators. Both [9] and [7] also propose to allow DAGs: this extension alone, as we have seen, ensures expressive completeness. But we will see below better justifications in terms of embeddability rather than succinctness.

When designing a FD language, is thus essential to have more than trees to reach expressive completeness. Trees, however, are easier to understand and manipulate because they have a compositional semantics. vDFD [22] manage to have both advantages.

6.3 Embeddability

An optional node n can be translated into a xor_2-node, say $n?$, with two sons: the original node n, and the TRUE node v which is an and_0-node (i.e., with no son). As we see in Fig.8, all incoming edges from parents of n are redirected to the new top node ($n?$), and all outgoing edges to sons start from the node n. This supports our view [13] that optionality is better treated as a decomposition operator (opt_1).

We constructed an embedding from OFD without constraints (called COFD in [13]) to VFD, presented in Table 2. To save space, we use the textual form for the graphs. For

[9] Operator arity is denoted by an underscript.

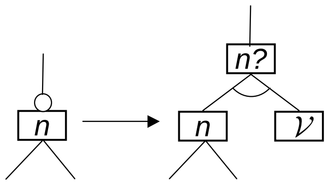


Fig. 8. Graphical embedding of redundant optional node (in OFD concrete syntax)

instance, a node bearing a xor_m operator is translated to a node bearing a $card_m[1\dots1]$ operator. In the next section, we will consider how those embeddings increase the size of the graph. Here we see that the VFD resulting from the embedding of a COFD diagram has the same size. This result indicates that *card*-nodes proposed by [7] can embed all the other constructs. We proposed thus to use them systematically inside tools. We slightly differ from [7] that also uses optional edges: these can be modelled by $card_1[0..1]$-nodes and would be harmfully redundant. We proposed VFD to eliminate this slight drawback. Please note that this latter suggestion only concerns abstract syntax. In the concrete syntax, it is probably a good idea to keep optional nodes as this would decrease the size and visual complexity of the diagrams.

Instead of ...	**write ...**
$opt_1(f)$	$card_1[0\dots1](f)$
$xor_m(f_1,\dots,f_m)$	$card_m[1\dots1](f_1,\dots,f_m)$
$and_s(f_1,\dots,f_s)$	$card_s[s\dots s](f_1,\dots,f_s)$

Table 2. Embedding COFD into VFD

6.4 Succinctness

When translations are not linear, it is still interesting to compute the increase in size of the graph, as measured by succinctness. RFD and OFD are of similar succinctness, but when translating VFD or EFD to OFD we translate a $card_k$-node to a OFD graph of size $O(k^2)$ [13]. A VFD of size $O(k)$ could contain k $card_k$-nodes, giving a cubic translation at the end: COFD $\leq O(VFD^3)$. This result indicates again that *card*-nodes are a useful addition, but for different reasons than presented in [7].

7 Discussion

The main limitation of our work is explicit in its scope: we address only *formal semantics*-related properties. In order not to over-interpret our conclusions, one should keep comprehensive view of model quality in mind. With respect to SEQUAL (Section 2), in order to be accurate and effective, we deliberately chose to address only part of the required qualities: *Domain appropriateness* is addressed by looking at language *expressiveness*. *Comprehensibility appropriateness* is addressed by looking at *embeddability* and *succinctness*. *Technical actor interpretation appropriateness* is addressed by looking at *complexity* and also *embeddability* and *succinctness*. Furthermore, our criteria cover only part of each of the three qualities. Future research should therefore devote similar attention to other qualities and criteria.

In contrast, a more holistic (quality-wise) attempt to compare FD languages is reported in [27]. It is specific though in the sense that it concerns the usage of FDs in a particular company, for a given kind of project. This leads us to point out that the notion of a "good" modelling language is only relative to the context of use of the language. The priorities to be put on the expected qualities and criteria are very likely to be different from one company, or projet, to another. This could lead us to relativise in some contexts the importance of formality. Still, we think that for FDs formality is very likely to deliver more than it will cost since (1) languages are relatively simple, (2) formality can be made largely transparent to the users (hidden behind a graphical concrete syntax), (3) the automation possibilities are many [13, 14, 28], and (4) correct FDs are mission-critical company assets that should suffer no ambiguity.

SEQUAL also helps identify another limitation: for now, we have only looked at *language* quality adopting a theoretical approach. A complementary work is to investigate models *empirically*. In Section 2, we emphasised the difficulty of such an endeavour because of the limited availability of "real" FDs. Nevertheless, we do not consider it impossible and can certainly learn a lot by observing how practitioners create and use FDs. Although we have focussed on studying theoretical properties of FD languages, we need to recognise that no formal semantics, nor criteria, can ever guarantee by itself that the languages help capture the right information (neither too little, nor too much) about the domain being modelled. Only empirical research can help us give a convincing answer to this other aspect of domain appropriateness.

A *threat to validity* is that all our reasoning (comparisons, demonstrations of theorems) was done by humans only, no tools. Human errors, miss- or over-interpretations are thus possible. Also, our formalisations were made only by considering the published documents, and without contacting the authors for clarifications, nor testing their tools. However, making $\mathcal{L}$, $\mathcal{S}$ and $\mathcal{M}$ explicit, we open the way for constructive discussion.

Finally, our method is yet to be applied to some relevant FD language proposals [16–21]. This is a prioritary topic of future work.

8 Conclusion

The bad news confirmed by this paper is that current research on variability modelling is fragmented. Existing research in the field is characterised by a growing number of proposals and a lack of accurate comparisons between them. In particular, the formal underpinnings of feature diagrams need more careful attention.

The nocuous consequences of this situation are: (1) the difficulty for practitioners to choose appropriate feature modelling techniques, (2) an increased risk of ambiguity in models, (3) underdeveloped, suboptimal or unsafe (i.e., not proved correct) tool support for reasoning on feature diagrams.

The good news that this paper delivers is that there are remedies to this situation. The ones that we propose are: (1) a global quality framework (e.g. Krogstie *et al.*'s SEQUAL) to serve as a roadmap for improving the quality of feature modelling techniques; (2) a set of formally defined criteria to assess the semantics-related qualities of feature diagram languages; (3) a systematic method to formalise these languages and make them ready for comparison and efficient tool automation; and (4) a first set of results obtained from the application of this systematic method on a substantial part of the feature modelling languages encountered in the literature.

Although the road ahead is still quite long, we are confident that the community can take profit of our proposal. It could be used for example as part of an arsenal to elaborate a standard feature modelling language. This standard would not suffer from ambiguity, and its formal properties (among others) would be well known, allowing to devise proved correct efficient reference algorithms. A similar approach could also be transposed to cognate areas where existing modelling techniques face similar challenges. In particular, we think of goal modelling techniques.

References

1. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
2. Cohen, S., Tekinerdogan, B., Czarnecki, K.: A case study on requirement specification: Driver Monitor. In: Workshop on Techniques for Exploiting Commonality Through Variability Management at the Second International Conference on Software Product Lines (SPLC2). (2002)
3. Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A feature-oriented reuse method with domain-specific reference architectures. Annals in Software Engineering **5** (1998) 143–168
4. Griss, M., Favaro, J., d'Alessandro, M.: Integrating Feature Modeling with the RSEB. In: Proceedings of the 5th International Conference on Software Reuse (ICSR'98), Vancouver, BC, Canada (1998) 76–85
5. Eisenecker, U.W., Czarnecki, K.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
6. Eriksson, M., Börstler, J., Borg, K.: The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations. In: Proceedings of the 9th International Conference on Software Product Lines (SPLC 2005). (2005) 33–44
7. Riebisch, M., Böllert, K., Streitferdt, D., Philippow, I.: Extending Feature Diagrams with UML Multiplicities. In: Proceedings of the Sixth Conference on Integrated Design and Process Technology (IDPT 2002), Pasadena, CA (2002)
8. Riebisch, M.: Towards a More Precise Definition of Feature Models. Position Paper. In: M. Riebisch, J. O. Coplien, D, Streitferdt (Eds.): Modelling Variability for Object-Oriented Product Lines (2003)
9. van Gurp, J., Bosch, J., Svahnberg, M.: On the Notion of Variability in Software Product Lines. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01). (2001)
10. Moody, D.L.: What Makes a Good Diagram? Improving the Cognitive Effectiveness of Diagrams in IS Development. In: Proceedings of the 15th international conference in Information Systems Development (ISD 2006). (2006)

11. Bontemps, Y., Heymans, P., Schobbens, P.Y., Trigaux, J.C.: Semantics of FODA Feature Diagrams. In Männistö, T., Bosch, J., eds.: Proceedings of Workshop on Software Variability Management for Product Derivation: Towards Tool Support, Boston (2004) 48–58
12. Bontemps, Y., Heymans, P., Schobbens, P.Y., Trigaux, J.C.: Generic Semantics of Feature Diagrams Variants. In: Proceedings of the 8th International Conference on Feature Interactions in Telecommunications and Software Systems(ICFI), IOS Press (2005) 58–77
13. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Generic semantics of feature diagrams. Computer Networks (2007) special issue on feature interactions in emerging application domains **51** (2007) 456–479
14. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Feature Diagrams: A Survey and A Formal Semantics. In: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06), Minneapolis, Minnesota, USA (2006) 139–148
15. Trigaux, J.C., Heymans, P., Schobbens, P.Y., Classen, A.: Comparative semantics of Feature Diagrams : FFD vs vDFD. In: Proceedings of Workshop on Comparative Evaluation in Requirements Engineering (CERE'06), Minneapolis, Minnesota, USA (2006)
16. Asikainen, T., Mannisto, T., Soininen, T.: A Unified Conceptual Foundation for Feature Modelling. In: Proceedings of the 10th International Software Product Line Conference. (2006) 31–40
17. Batory, D.S.: Feature Models, Grammars, and Propositional Formulas. In: Proceedings of the 9th International Conference on Software Product Lines (SPLC 2005). (2005) 7–20
18. Benavides, D., Ruiz-Cortés, A., Trinidad, P.: Automated Reasoning on Feature Models. LNCS, Advanced Information Systems Engineering: Proceedings of the 17th International Conference, CAiSE 2005 **3520** (2005) 491–503
19. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing Cardinality-based Feature Models and their Specialization. Software Process: Improvement and Practice **10** (2005) 7–29
20. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration Using Feature Models. Software Process Improvement and Practice, special issue on Software Variability: Process and Management **10** (2005) 143 – 169
21. Sun, J., Zhang, H., Li, Y.F., Wang, H.: Formal Semantics and Verification for Feature Modeling. In: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2005. (2005) 303–312
22. van Deursen, A., Klint, P.: Domain-Specific Language Design Requires Feature Descriptions. Journal of Computing and Information Technology **10** (2002) 1–17
23. Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"? IEEE Computer **37** (2004) 64–72
24. Krogstie, J.: Using a semiotic framework to evaluate UML for the development of models of high quality. Unified Modeling Language: System Analysis, Design and Develoment Issues, IDEA Group Publishing (2001) 89–106
25. Krogstie, J., Sindre, G., Jørgensen, H.: Process Models Representing Knowledge for Action: a Revised Quality Framework. Eur. J. Inf. Syst. **15** (2006) 91–102
26. Heymans, P., Schobbens, P.Y., Trigaux, J.C., Matulevičius, R., Bontemps, Y., Classen, A.: Evaluating Formal Properties of Feature Diagrams. Technical report, University of Namur (2006)
27. Djebbi, O., Salinesi, C.: Criteria for Comparing Requirements Variability Modeling Notations for Product Lines. Workshop on Comparative Evaluation in Requirements Engineering (CERE'06) **0** (2006) 20–35
28. Benavides, D., Ruiz-Cortés, A., Trinidad, P., Segura., S.: A Survey on the Automated Analyses of Feture Models. In: Jornadas de Ingeniería del Software y Bases de Datos (JISBD). (2006)

Ontology-Based Software Reliability Modelling

Jiehan Zhou, Eila Niemelä, Antti Evesti

VTT Technical Research Centre of Finland
Kaitoväylä 1, 90571 Oulu, Finland
Email:{firstname.surname}@vtt.fi

Abstract. Reliability has become a major concern for software-intensive systems. This paper proposes a novel ontology-based method for software reliability modelling, including a software reliability ontology and an ontology-based software reliability modelling system. The software reliability ontology is analysed and developed for software reliability engineering with respect to domains of reliability measurement processes, methods, models, organization, and tools. The ontology-based software reliability modelling system validates the ontology-based method with presentations of the system infrastructure, the system implementation techniques, and the system application cases.

1. Introduction

Modern society is permeated by software-intensive systems, such as consumer electronics, buildings, automobiles, and aircraft. The size and complexity of software-intensive systems have grown dramatically during the past decade, and the trend is certain to continue with the emergence of service-oriented architectures and Web services [1, 2]. Meanwhile, software reliability has become a major concern for software-intensive systems.

'Software reliability' refers to the probability of failure-free operation of a computer program in a specified environment over a specified period of time [3-5]. In the interest of increasing software reliability, a number of studies have been carried out on reliability modelling [6-8], reliability estimation, and prediction tools development [9, 10]. At the same time, several books [4, 5, 11] have been published for the purposes of reliability education and training. For the sake of brevity, when speaking of 'reliability' below, we are referring to software reliability.

In what follows, we will explore a novel ontology-based method for reliability modelling, including a reliability ontology and an ontology-based reliability modelling system. To the best of our knowledge, there are only few reports available on the topic. The remainder of the paper is organized as follows: Section 2 outlines a set of key concepts related to ontology-based reliability modelling. Section 3 defines the objectives of building a reliability ontology. In Section 4, different ontology engineering methods are compared and a guideline is presented for reliability ontology modelling. Section 5 examines reliability domains and develops the reliability ontology. The reliability modelling system is presented in Section 6, along with the system infrastructure, system implementation techniques, and some system application cases. Section 7 draws the conclusions from our discussion and anticipates directions for future research.

2. Concepts

Ontology is a shared knowledge standard or knowledge model explicitly defining primitive concepts, relations, rules, and their instances. Ontology can be used for capturing, structuring, and enlarging explicit and tacit knowledge across people, organizations, and computer and software systems [12].

Reliability ontology consists of concepts and their relationships related to the topic of reliability engineering and aimed at facilitating the work of reliability experts in managing and developing reliability knowledge.

Ontology-based reliability design is an ontology-based method that provides reliability experts with the reliability ontology and associated management tools for facilitating software reliability definition and measurement.

3. Objectives of the reliability ontology development

The intended uses of the reliability ontology include the following application contexts:

§ Management and development of reliability knowledge. By making use of the reliability ontology, the software customer can understand reliability terminologies explicitly, assess the reliability of the provided software or service, and fluently communicate with reliability experts. The reliability experts can adopt, adjust, and choose reliability models based on the reliability ontology.

§ Support for computer-aided reliability estimation and prediction. Once the reliability ontology is built into software programs (so-called ontology-based programs), these programs can be enhanced by customizing reliability measurement procedures and by providing a common reliability development and management framework.

§ Support for reliability knowledge management in software engineering. Component and service reliability becomes a critical factor influencing component-based and service-oriented software development. The reliability ontology promises to provide software community with a common communication platform for addressing reliability knowledge, and to facilitate reliability-based service discoveries and compositions.

4. Ontology engineering and reliability ontology modelling

4.1 Ontology engineering methodologies

The ontology engineering methods are summarised in Table 1. An ontology engineering method is a series of interrelated activities and techniques used for creating ontology. Thus, the traditional ontology development methods given in Table 1 are presented in terms of involved activities, created ontologies, and unique features. A more detailed comparison between the different ontology engineering methodologies is presented in [13].

Table 1. Summary of traditional ontology development methods

Method	Process	Ontologies created	Notes
Cyc [14]	Manual coding, computer coding, computer managing	Cyc	Unanimous knowledge capture
Uschold and King [15]	Purpose identification, ontology building, evaluation, and documentation	Enterprise ontology	Relevant to business enterprise
Gruninger and Fox [16]	Scenario identification, informal competency question formulation, terminology specification, formal competency question formulation, axiom specification, and completeness theorem specification	TOVE ontologies including enterprise design, project, scheduling, and service ontology	High degree of formality
KACTUS [17]	Application specification, preliminary design, refinement, and structuring	Fault and service recovery planning ontology	Conditioned by application development
Methontology [5, 18][8]	Roots in IEEE1074-1995	Chemicals, reference, KM ontology, etc.	Most mature, the method proposed by FIPA
SENSUS-based [19]	Identify seed terms, link the seed terms to SENSUS, add paths to the root, add new domain terms, and add complete sub-trees	Military air campaign planning	Easy generation of skeleton ontologies from huge ontologies
On-To-Knowledge [20, 21]	Feasibility study, kick-off, refinement, evaluation, and maintenance	Skills management, virtual enterprise, OntoWeb, etc.	Created ontologies highly dependent on the application, ontology learning

4.2 Reliability ontology modelling

Based on the summary of ontology engineering methods and a typical ontology engineering process given in [8], we can propose a guideline for creating the reliability ontology:

Step 1. Determine the domain of the reliability ontology. The reliability ontology covers reliability process, method, model, specification, tool, and organization domains.

Step 2. Consider reusing existing reliability ontologies. Unfortunately, no reliability ontologies exist as of yet. Information on reliability is distributed though books and research publications.

Step 3. Enumerate important reliability concepts. It is useful to start with classical works on reliability engineering, creating a list of all the concepts that reliability experts would prefer to use.

Step 4. Define the reliability concept hierarchy. A top-down development process may be used in this step. First we define the most general reliability concepts (process, method, and specification) and the subsequent specialisation of these concepts (reliability definition and operational profile development).

Step 5. Define the reliability properties. In general, there are two types of reliability properties: internal properties and external properties. The internal properties indicate properties belonging to the concept itself, such as the name of a process. The external properties indicate relationships between concepts, such as the method of a process.

Step 6. Create reliability instances. We define an individual reliability instance in terms of a concept and its properties.

5. Reliability ontology design

5.1. Reliability ontology domains

We describe reliability engineering as a series of interrelated processes by which reliability knowledge is reorganized with the support of methods, tools, models, organization, and the specifications of input and output (Fig. 1).

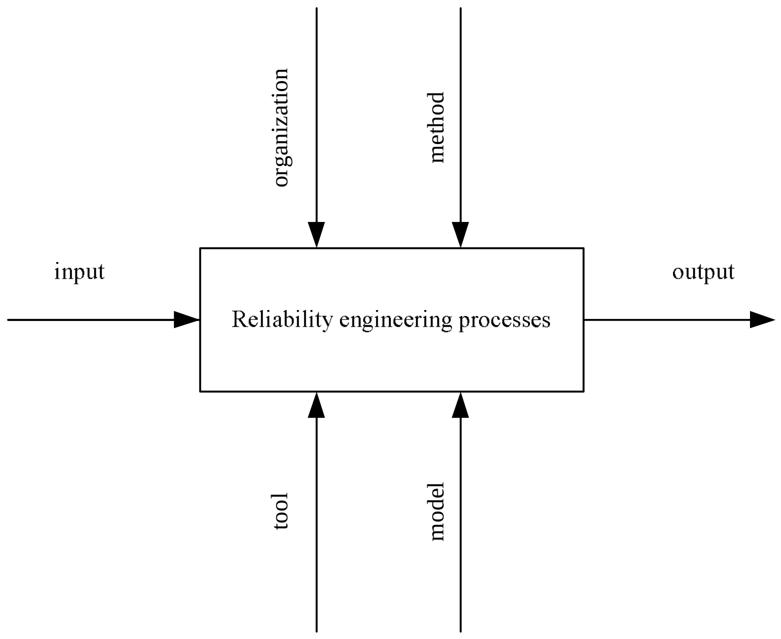


Fig. 1. Reliability ontology domains

The reliability process refers to reliability engineering processes, consisting of five activities: a) definition of necessary reliability, b) development of operational profiles, c) preparation for test, d) execution of test, and e) application of failure data to guide decision-making.

The reliability organization is in charge of executing the processes. Reliability measurement is usually carried out in an operation organization. The commonly involved roles are the end-user for executing the software, the manager for resource allocation, the system engineers for tailoring reliability procedures, and the quality assurance engineers for running tools and collecting failure data.

The reliability method identifies a way for the organization to undertake reliability processes in terms of cost-efficiency and purpose-specification.

The reliability models are chosen and used in any given applications depending on the purpose of the application. A software reliability model usually has the form of a random process that describes the behaviour of failures over time.

Reliability tools are computer programs or simulations used in software reliability measurement.

Reliability specification of input and output refers to the input and output data for the reliability process. This data can take the form of tables, files, and graphics.

5.2. Concepts hierarchy

A reliability concepts hierarchy is a hierarchical concept classification. Fig. 2 describes these concepts in detail.

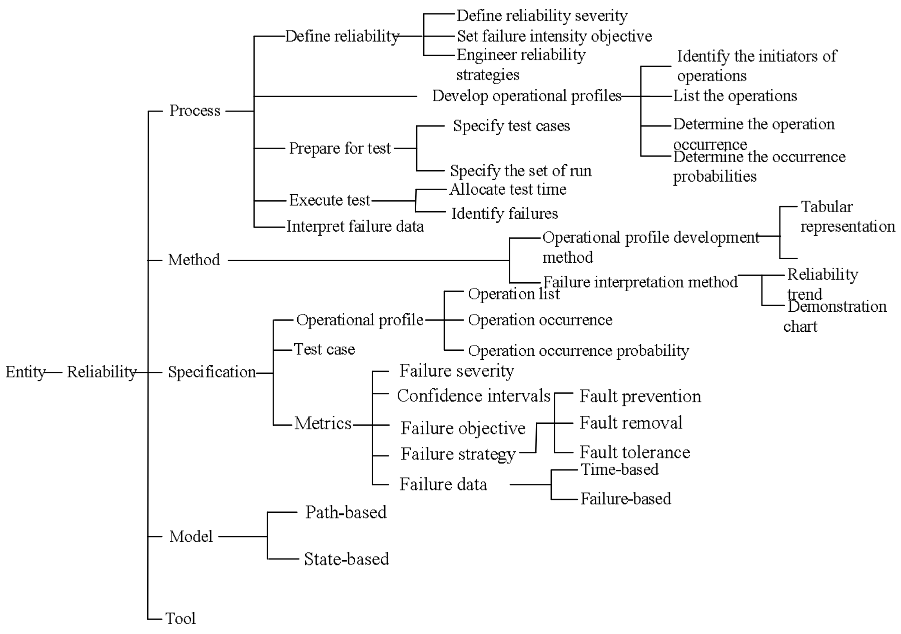


Fig. 2. Reliability concept hierarchy

Reliability process concepts

Defining reliability means quantitative reliability definition of a software system, which in turn makes it possible for reliability experts to balance customer needs for reliability, delivery date, and cost. The reliability definition mainly consists of the sub-processes of reliability severity definition, failure intensity objective setting, and reliability strategies engineering.

The operational profile development mainly consists of the sub-processes of identifying the initiator of operations, listing the operations, determining the operation occurrence, and determining the occurrence probabilities.

In preparing for test, the operational profile information is applied to planning efficient testing, including preparation of test cases and test procedures.

In executing the test, the test cases and test procedures are used in such a way that the desired goal of the efficient test can be achieved. The execution of tests involves three main activities: allocating test time, invoking the tests, and identifying failures that occur.

In failure data interpretation, the predicted reliability is compared with the set reliability objectives for guiding system-reliability decision making, e.g., accepting or rejecting an acquired component or a software system.

Reliability method concepts

A method describes how to conduct a process efficiently and effectively. The reliability method mainly consists of the methods used in the reliability processes.

The following lists the methods that are presented in [5], including operational profile development methods and failure interpretation methods.

Operational profile development methods include tabular representation and graphical representation. Tabular representation generally works better for systems whose operations have few (often only one) attributes. Graphical representation is generally better suited for software the operations of which have multiple attributes.

Failure interpretation methods. There are two approaches for interpreting software system reliability: failure intensity trend and reliability chart demonstration. The failure intensity trend method estimates the failure intensity over all severity classes and across all operational modes against time. The reliability chart method is used in certification tests in which each failure is plotted and labelled with its severity class.

Specification concepts

Specifications are the input or output of the reliability measurement processes, such as operational profiles, test cases, and reliability metrics.

An operational profile is an operation scenario that describes typical uses of the system, consisting of an operation list, the operation occurrence, and the operation occurrence probability. The operation list is a tabular or graphical representation of the operations each initiator produces. The operation occurrence refers to the occurrence rates of the operation. The operation occurrence is commonly measured with respect to time. The operation occurrence probability is the ratio of the operation compared to the total occurrence rates.

A test case is the partial specification of a run through the naming of its direct input variables and the values of these variables during the preparations for the test process.

Reliability metrics are measurable, quantitative software attributes, consisting mainly of the following items:

- A failure is the departure of program behaviour from user requirements during execution. A failure confidence interval represents a range of values within which a parameter is expected to lie with a certain statistical degree of probability.
- A failure severity class is a set of failures that affect users to the same degree or level. The severity is often related to the criticality of the operation that fails [5, 11].
- A failure objective is set for the software system.
- Failure strategies include fault prevention, fault removal, and fault tolerance. Fault prevention uses requirements, design, and coding technologies and processes to reduce the number of faults. Fault removal uses code inspection and development testing to remove faults in the code once it is written. Fault tolerance reduces the number of failures that occur, by detecting and countering deviations in program execution that may lead to failures [5].
- The failure data refers to the metric of representing failure occurrence, including the time-based class and the failure-based class. The failure-based class represents failure occurrence by indicating the frequency of the failures experienced within a time interval. The time-based class represents failure occurrence by determining the time interval between failures.

Model concepts

A reliability model usually has the form of a failure process that describes the behaviour of failures with time, also called failure random process. The possibilities for different mathematical forms to describe the failure process are almost limitless [5]. The following lists two classes of reliability models used in software architecture reliability evaluation.

The state-based models use a control flow graph to represent the architecture of the system. It is assumed that the transfer of control between components has a Markov property, which means that, given what is known of the component in control at any given time, the future behaviour of the system is conditionally independent of its past behaviour [22]. State-based models consist of the states, or externally visible modes of operation, that must be maintained, and the state transitions labelled with system inputs and transition probabilities. State-based models can be used even if the source code of the component is not available. State-based model instances include: the Littlewood model [23], the Cheung model [24], the Laprie method [25], the Kubat method [26], the Gokhale et al. method [27], and the Ledoux method [28].

The path-based models are based on the same common steps as the state-based models [22]. In addition, the path-based models enable one to specify, with the help of the simulation, component reliability estimations [29]. Path-based model instances include: the Shooman model [30], the Krishnamurthy and Mathur model [31], and the Yacoub et al. model [32].

Tool concepts

This section presents some reliability evaluation tool instances, such as SMERFS [4], CASRE[9], SoRel [33], and AgenaRisk [34].

SMERFS (Statistical Modelling and Estimation of Reliability Functions for Systems) [4] allows the end user to enter data, to edit and/or transform the data if necessary, to plot the data, to select an appropriate model to fit the data, to determine the fit of the model using both statistical and graphical techniques, to make various reliability predictions based upon the fitted model, and to try different models if the initial model proves inadequate.

CASRE (Computer Aided Software Reliability Estimation) [9, 35] is an extension of SMERFS. Users are guided through the selection of a set of failure data and the execution of a model with the assistance of selectively enabling pull-down menu options.

SoRel [33] is a tool for software (and hardware) reliability analysis and prediction that provides qualitative and quantitative elements concerning, for instance, a) the evolution of the reliability in response to the debugging effort; b) the estimation of the number of failures for the subsequent time periods to allow test effort planning and the assignment of the numerical importance by the test and/or maintenance team; and c) the prediction of reliability such as the mean time to failure, the failure rate, and the failure intensity.

AgenaRisk [34] is a risk assessment and risk analysis tool. It arms users with the latest algorithms that allow quantification of uncertainty and offer models for prediction, estimation, and diagnosis, all made accessible via a sophisticated graphical user interface.

Additional software reliability modelling tools and programs are surveyed in [36] and listed on the Web [37].

5.3. Properties definition

We categorize the reliability property into internal property and external property. The internal property describes the internal structure and attributes of concepts, and the external property describes the relationships between concepts. For an internal property, we must determine which concept it describes; for instance, specificationName is one of the internal properties of the concept Specification. For an external property, we must determine the class(es) of which the values of the property will be members, and the class(es) that will have the property as a member; for instance, hasMethod is an internal property between concepts of Method and Process. The initial reliability properties are defined in Fig. 3.

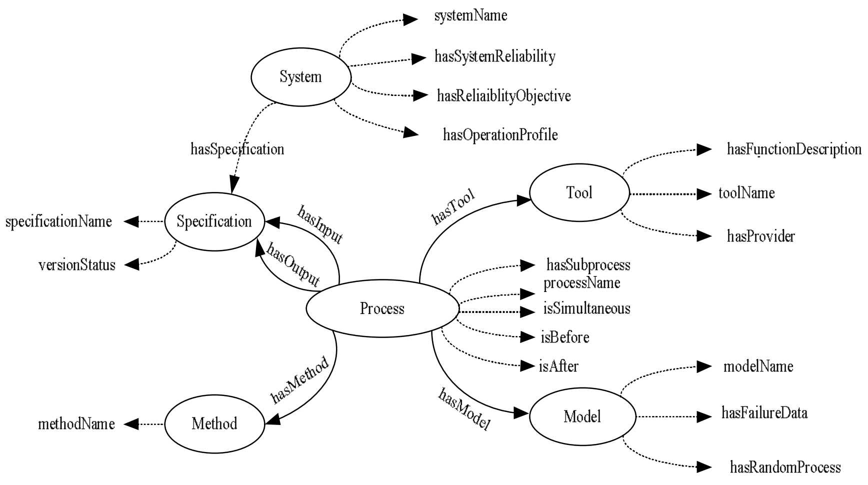


Fig. 3. Reliability properties definition

System properties

systemName: a system has a name.
hasSystemReliability: a system has a reliability objective.
hasOperationProfile: a system has an operational profile.
hasReliabilityObjective: a system has a failure intensity objective or reliability objective.
hasSpecification: a system has at least one specification.

Process properties

processName: a process has a name.
hasSubprocess: a process could have two or more sub-processes.
isBefore/isAfter/isSimutaneous: a process can occur before/after/simultaneously with at least one other process.
hasMethod: a process could have a method.
hasTool: a process could have a supporting tool.
hasModel: a process could have a model.
hasInput: a process could have an input specification.
hasOutput: a process could have an output specification.

Tool Properties

toolName: a tool must have a name.
hasFunctionalDescription: a tool must have a functional description.
hasProvider: a tool must have a provider.

Model properties

modelName: a model must have a name.
hasFailureData: a model must have failure data for input.
hasRandomProcess: a model must have a random process.

Specification properties

specificationName: a specification must have a name.
versionStatus: a specification has a version status.

Method Properties

methodName: a method must have a name.

6. Ontology-based reliability modelling system

6.1. System infrastructure

Suppose a company is planning to introduce a new software product of HomeSecurity for monitoring home environment information automatically. The system consists of a set of home automation service components (e.g. a video capture service, a light switch service, and an alarming service). Those components must satisfy quality requirements to some degree. For example, the alarm service needs a high reliability in the HomeSecurity system. In order to facilitate product reliability measurement, we have designed an ontology-based reliability modelling system, which has the primary functions of managing the software reliability ontology and designing software system reliability (Fig. 4).

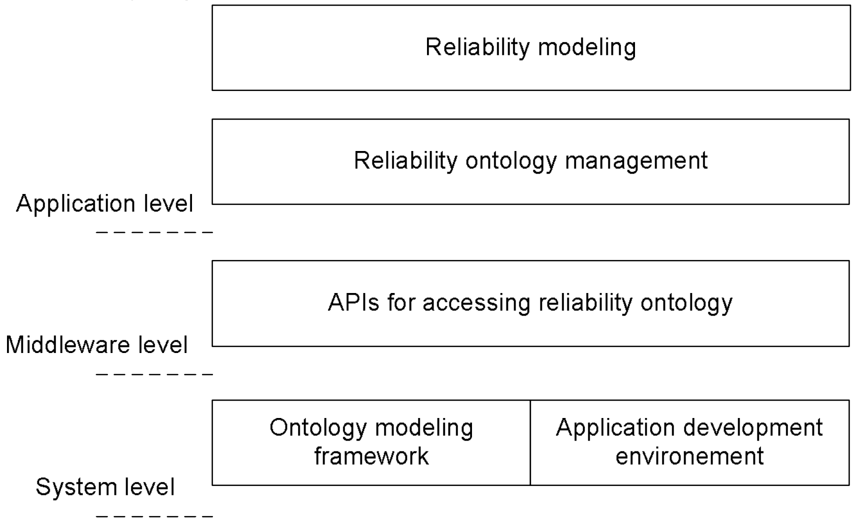

Reliability modeling
Reliability ontology management
Application level
APIs for accessing reliability ontology
Middleware level
Ontology modeling framework
Application development environement
System level

Fig. 4. Ontology-based reliability modelling system

The system level consists of an ontology modelling framework and an application development environment. The ontology modelling framework is used for reliability ontology modelling by allowing reliability experts to create reliability ontology and load reliability ontology files. The application development environment is an application development platform used for building, deploying, and managing application software across the lifecycle.

The middleware level consists of APIs (Application Programming Interfaces) responsible for handling reliability ontology, including build, access, display, and update reliability ontology.

The application level includes reliability ontology management and reliability modelling. The reliability ontology management application is responsible for reading, writing, and visualizing reliability ontology documents written in Web ontology languages (e.g., RDF [38], and OWL [39]). The reliability ontology management further enables reliability experts to make semantic knowledge queries about reliability. The reliability modelling application supports software system reliability measurement processes, including definition of reliability, development of operation profiles, preparation for and execution of reliability test, and interpretation of failure data.

6.2. System implementation

In the implementation of the ontology-based reliability modelling system, we have explored and adopted the following technonologies: Eclipse for the application development environment, OWL for the reliability ontology modelling, and Jena for the reliability ontology file management.

§ Eclipse for the ontology-based system development environment [40]. Eclipse is an open source development platform comprised of extensible frameworks and tools for building, deploying, and managing software across the lifecycle. Eclipse possesses a powerful modelling framework and code generation facility (Eclipse Modelling Framework, or EMF). The EMF provides the foundation for interoperability with other EMF-based tools and applications.

§ OWL (Web Ontology Language) [41] for the reliability ontology modelling. OWL enables greater machine interpretability of Web content than XML, RDF, or RDF Schema. OWL further provides additional vocabulary along with formal semantics, and adds qualifiers for describing properties and classes, such as disjointness, cardinality, and symmetry. The reliability OWL documents refer to files written with the OWL language.

§ Jena API [42] for accessing reliability ontology OWL documents. Jena framework provides rich OWL APIs for reading and writing reliability ontology OWL documents. These can be used to save and read the reliability ontology in the form of files.

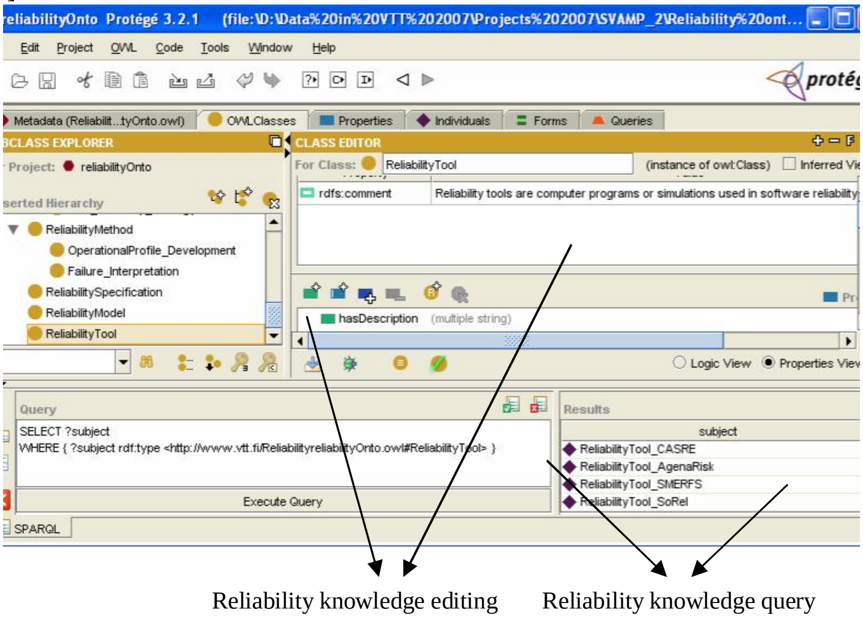
6.3. Application cases

To demonstrate the usefulness of the ontology-based reliability modelling system, including reliability ontology document management, reliability ontology query and reliability modelling, we would like to briefly mention some application cases supported by the system.

§ Reliability ontology document management. In this case, the ontology-based design environment enables reliability experts to load reliability ontologies from and to OWL files and graphically create, modify, and store the reliability ontology. At the moment, we adopt Protégé [43] to fulfill reliability ontology management (See Figure 5 (a)).

§ Reliability ontology query. In this case, the ontology-based modelling system enables experts to make semantic reliability knowledge queries through a friendly user interface, for example, Q1 (internal reliability property query): What tool instances does the class of 'reliability tool' have? Q2 (external reliability properties query): What is the output for the process of 'operational profile development' when choosing the 'tabular representation' method? (See Figure 5 (a))

§ Reliability modelling. In this case, the system enables reliability experts to conduct system reliability measurements, to define software system reliability objectives, to develop system operational profiles, to prepare and execute system reliability tests, and to interpret system reliability. Figure 5 (b) presents the user interface of the Quality Profiler. The Quality Profiler utilizes the reliability ontology and allows the end user to specify quality requirements for a given software component. The detail about the Quality Profiler module is given in [44].

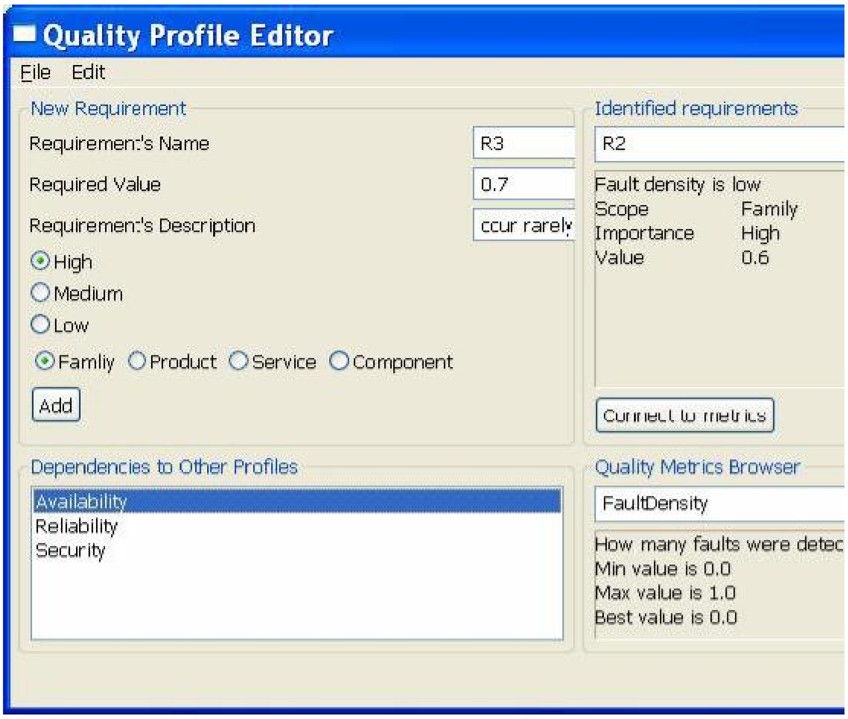

eliabilityOnto Protégé 3.2.1 (file:\D:\Data%20in%20VTT%202007\Projects%202007\SVAMP_2\Reliability%20ont...
Edit Project OWL Code Tools Window Help
protég
Metadata (Reliabilit...tyOnto.owl)
OWLClasses
Properties
Individuals
Forms
Queries
BCLASS EXPLORER
CLASS EDITOR
Project: reliabilityOnto
For Class: ReliabilityTool
(instance of owl:Class)
Inferred Vi
serted Hierarchy
ReliabilityMethod
OperationalProfile_Development
Failure_Interpretation
ReliabilitySpecification
ReliabilityModel
ReliabilityTool
rdfs:comment
hasDescription (multiple string)
Logic View
Properties View
Query
Results
Execute Query
SPARQL
Reliability knowledge editing
Reliability knowledge query

Reliability tools are computer programs or simulations used in software reliability

```
SELECT ?subject
WHERE { ?subject rdf:type <http://www.vtt.fi/ReliabilityreliabilityOnto.owl#ReliabilityTool> }
```

subject
ReliabilityTool_CASRE
ReliabilityTool_AgenaRisk
ReliabilityTool_SMERFS
ReliabilityTool_SoRel

a. Reliability ontology management and ontology-based reliability knowledge query



Requiremen:'s Name R3
Required Value 0.7
Requiremen:'s Description ccur rarely
High
Medium
Low
Famliy Product Service Component

R2
Fault density is low
Scope Family
Importance High
Value 0.6

Availability
Reliability
Security

FaultDensity
How many faults were detec
Min value is 0.0
Max value is 1.0
Best value is 0.0

b. Ontology-based reliability-aware software modelling
Fig.5. Reliability ontology-based applications

7. Conclusions and future work

Growing attention has been given to the quality driven software design, increasing software complexity and emerging service-oriented architectures. At present, only few studies exist on ontology-based software reliability design. In the foregoing we have explored and proposed a novel ontology-based method for designing software reliability. The ontology-based method aims to provide reliability experts with a reliability ontology and related computer-aided tools for facilitating reliability engineering. First, the concepts related to reliability were specified. Next, studies associated with ontology engineering were discussed, including the objectives of creating a reliability ontology, ontology engineering methods, and guidelines for creating a reliability ontology. Further, by identifying the knowledge scopes of reliability-aware software design, the reliability ontology was designed primarily with respect to reliability concepts and properties. The experiences gained in developing the ontology-based reliability design tool were also presented. Future work will focus on elaborating the ontology-based method in the following aspects:

§ Continuing the development of reliability ontology, along with the mining and refining of the reliability concepts and properties.

- § Applying the method to software architecture design. This work will extend the existing reliability ontology by developing and merging a software architecture ontology and developing associated applications supporting reliability-aware software architecture design.
- § Elaborating the implementation of the ontology-based reliability modelling system for specified technical features.

References

1. Kreger, H.: Web Service Conceptual Architecture (WSCA 1.0). www-3.ibm.com/software/solutions/webservices/pdf/WSCA.pdf, 2001.
2. WSA/W3C.: Web Services Architecture. http://www.w3.org/TR/ws-arch/#whatis, 2004.
3. Standard Glossary of Software Engineering Terminology. ANSI/IEEE 1991.
4. Lyu, M. R.: Handbook of software reliability engineering. McGraw-Hill, 1995.
5. Musa, J.: Software reliability engineering, more reliable software faster development and testing: McGraw-Hill, 1998.
6. Wallace, D. R.: Practical software reliability modeling. 26th Annual NASA, Software Engineering Workshop, 2001.
7. Wang, W.L., Chen, M.H.: Heterogeneous software reliability modeling. 13th International Symposium on Software Reliability Engineering, 2002.
8. Noy, N. F., McGuinness, D. L.: Ontology Development 101: A Guide to Creating Your First Ontology. http://ksl.stanford.edu/people/dlm/papers/ontology101/ontology101-noy-mcguinness.html, 2006.
9. Lyu, M. R., Nikora, A.: CASRE: a computer-aided software reliability estimation tool. Fifth International Workshop on Computer-Aided Software Engineering, 1992.
10. Ramani, S., Gokhale, S. S., Trivedi, K. S.: SREPT: software reliability estimation and prediction tool. Performance Evaluation, vol. 39, pp. 37 - 60, 2000.
11. Musa, J. D., Iannino, A., Okumoto, K.: software reliability: measurement, prediction, application: McGraw-Hill Book Company, 1987.
12. Zhou, J.: Knowledge Dichotomy and Semantic Knowledge Management. In the proceedings of the 1st IFIP WG 12.5 working conference on Industrial Applications of Semantic Web, Jyvaskyla, Finland, 2005.
13. Zhou, J., Niemela, E.: State of the Art on Metamodel-Driven Multimedia over Mobile Ubiquitous Computing Environments. 4th IASTED International Conference on Communications, Internet and Information Technology CIIT2005, Cambridge, USA, 2005.
14. Lenat, D. B., Guha, R.V.: Building large knowledge-based systems: representation and inference in the Cyc project. Boston, Massachusetts: Addison-Wesley, 1990.
15. Uschold, M., King, M.: Towards a methodology for building ontologies. IJCAI'95 Workshop on Basic Ontological Issues in Knowledge Sharing, Montreal, Canada, 1995.
16. Gruninger, M., Fox, M. S.: Methodology for the design and evaluation of Ontologies. IJCAI'95 Workshop on Basic Ontological Issues in Knowledge Sharing, Montreal, Canada, 1995.
17. Bernaras, A., Laresgoiti, I., Corera, J.: Building and reusing ontologies for electrical network applications. European Conference on Artificial Interlligence (ECAI'96), Budapest, Hungary, 1996.
18. Computer Society. IEEE standard for developing software life cycle processes. IEEE std 1074. New York, 1995.

19. Swartout, B., Patil, R., Knight, K., Russ, T.: Toward distributed use of large-scale ontologies. AAAI'97 Spring Symposium on Ontological Engineering, Standford University, California, 1997.
20. Staab, S., Schnurr, H. P., Studer, R., Sure, Y.: Knowledge processes and ontologies. IEEE Intelligent Systems, vol. 16, pp. 26-34, 2001.
21. Roshandel, R., Medvidovic, N.: Toward Architecture-based Reliability Estimation. The International Conference on Dependable Systems and Networks, Florence, Italy, 2004.
22. Popstojanova, K. G., Trivedi, K. S.: Architecture-based approach to reliability assessment of software systems. Performance Evaluation, vol. 45, pp. 179-204, 2001.
23. Littlewood, B.: Software reliability model for modular program structure. IEEE Trans. Reliability, vol. 28, pp. 241-246, 1979.
24. Cheung, R. C.: A user-oriented software reliability model. IEEE Trans. Software Eng., vol. 6, pp. 118-125, 1980.
25. Laprie, J. C.: Dependability evaluation of software systems in operation. IEEE Trans. Software Eng., vol. 10, pp. 701-714, 1984.
26. Kubat, P.: Assessing reliability of modular software. Operation Research Letters, vol. 8, pp. 35-41, 1989.
27. Gokhale, S., Wong, W. E., Trivedi, K., Horgan, J. R.: An analytical approach to architecture based software reliability prediction. Third International Computer Performance and Dependability Symposium (IPDS'98), 1998.
28. Ledoux, J.: Availability modeling of modular software. IEEE Trans. Reliability, vol. 48, pp. 159-168, 1999.
29. Immonen, A.: A method for predicting reliability and availability at the architectural level. in Research Issues in Software Product-Lines - Engineering and Managemen, T. Kakola and J. C. Duenas, Eds. Berlin Heidelberg: Springer Verlag, 2006, pp. 373-422.
30. Shooman, M.: Structural models for software reliability prediction. Second International Conference on Software Engineering, 1976.
31. Krishnamurthy, S. Mathur, A. P.: On the estimation of reliability of a software system using reliabilities of its components. Eighth International Symposium on Software Reliability Engineering (ISSRE?7), 1997.
32. Yacoub, S., Cukic, B., Ammar, H.: Scenario-based reliability analysis of component-based software. 10th International Symposium on Software Reliability Engineering (ISSRE'99), 1999.
33. Kanoun, K., Kaaniche, M., Laprie, J.-C., Metge, S.: SoRel: A tool for reliability growth analysis and prediction from statistical failure data. The Twenty-Third International Symposium on Fault-Tolerant Computing, 1993.
34. AgenaRisk. http://www.agenarisk.com/, 2006.
35. AT&T SRE Toolkit. http://www.cse.cuhk.edu.hk/~lyu/book/reliability/sretools.html, 2006.
36. Stark, G. E.:A Survey of Software Reliability Measurement Tools. The International Symposium on Software Reliability Engineering, 1991.
37. Reliability Modeling Programs.http://www.enre.umd.edu/tools/rmp.htm
38. RDF/W3C.: Resource Description Framework (RDF). http://www.w3.org/RDF/, 2005.
39. W3C-OWL.: OWL Web Ontology Language Overview. 2005.
40. Eclipse. http://www.eclipse.org/, 2006.
41. OWL Web Ontology Language Overview. http://www.w3.org/TR/2004/REC-owl-features-20040210/#s1.2, 2005.
42. Jena A Semantic Web Framework for Java. http://jena.sourceforge.net/index.html, 2005.
43. Protégé. http://protege.stanford.edu/.
44. Evesti, A.: Quality oriented software architecture development. Department of Electrical and Information Engineering. Oulu: University of Oulu, 2007, pp. 57.

An Analysis of Variability Modeling and Management Tools for Product Line Development

Rafael Capilla[1], Alejandro Sánchez[1], Juan C. Dueñas[2*]

[1] Department of Computer Science, Universidad Rey Juan Carlos
c/ Tulipán s/n, 28933, Madrid, Spain
rafael.capilla@urjc.es, valdezate@gmail.com
[2] Department of Engineering of Telematic Systems, ETSI Telecomunicación
Ciudad Universitaria s/n, 28040, Madrid, Spain
jcduenas@dit.upm.es

Abstract. Software variability is considered a key technique for modelling the variable parts of a software system. The importance for representing this variability is well recognized in product lines architectures. The advantage for producing multiple products under strong time to market conditions is reached through the definition of appropriate variation points as configurable options of the system. In industry, the development of large software systems requires an enormous effort for modelling and configuring multiple products. To face this problem, variability modelling tools have been proposed and used for representing and managing the variations of a system. In this work we analyze the current state of practice of some of these tools to identify their capabilities and limits and we provide suggestions for future second-generation tools.

1. Introduction

Software architectures have been widely used for almost 25 years for representing the main parts of a software system [2]. The success of software architectures comes from their ability for representing the common and variable parts of a set of related systems. The aim of software variability is to exploit the variable parts of a set of systems and to configure new products from the same family. Software variability has been widely used during the last years in product line architectures [6] for building multitude of products as a way to meet the market condition when multiple products have to be delivered in short time. Variation points are used to represent this variability as well as to discriminate and to configure different products. One of the main advantages is that variation points are used to delay the design decisions made at the beginning to the latest moment in the product lifecycle. Different alternatives for representing and configuring these variation points are possible, but modeling and managing hundreds of variation points constitutes a current problem, because dependencies between these have to be handled. The remainder of this paper is as fol-

[*] The work performed by Juan C. Dueñas has been partially done in the ITEA-SERIOUS project, under grant FIT340000-2005-136 of Ministerio de Industria, Turismo y Comercio de España.

lows. Section 2 describes the main concepts of software variability. Section 3 outlines variability representation and visualization techniques. Section 4 describes the main characteristics of a representative list of variability modeling tools. Section 5 mentions the impact of variability modeling tools in product lines. Section 6 discusses the limits of current tools and the features that might be implemented for future tools.

2. Software Variability Concepts

In order to analyze and understand the features implemented in the tools examined in this work, in this section we provide an overview of the main concepts belonging to software variability.

2.1 Origins

At the beginning of the '90s, the Feature-Oriented Domain Analysis (FODA) method [16] was proposed for modeling the variations of software systems. FODA defines mandatory, alternative and optional features and composition rules for guiding the rationale in which the visible and external properties of systems are represented. Some approaches improve FODA capabilities. The Feature-Oriented Reuse Method (FORM) [17] extends FODA with *domain features* (i.e.: features specific to a particular domain) and performs a classification of different types of features. FORM is a systematic method that captures the commonalities and differences of application in a domain in terms of features and produces different feasible configurations of reusable architectures. Indeed, these feature trees are used as a "decision model" in the engineering process to obtain different product configurations. In [8] the authors propose to include quantitative values to FODA trees for specifying QoS parameters in distributed systems. The definition of quantitative values and ranges values among are quite useful for featuring certain applications (e.g. telecommunication systems). Today, the success of software product lines in the industry has raised the popularity of feature trees like FODA for describing the variability of the product family. A FODA tree describes at a high level (i.e.: conceptual level) the variations and alternatives that occur in a particular software product or in the entire product family as specified in the requirements. This high level description must be translated to the design and implementation levels in the form of variation points and variants.

2.2 Variation Points and Variants

The concrete specification of a feature tree is usually achieved by means of variation points (VP). We understand by variation points an area of a software system affected by variability. Each variation point is commonly defined in terms of variants that represent the alternatives for each variation point. At the end, each variant may define a set of allowed values that are selected at a given time. Variability *in space* defined the allowed configurations for a set of products. Furthermore, the extensibility of a variation point can be *open* or *closed*. In closed variation points all variants are defined at pre-deployment time and the selection of the choices is only possible between the built-in variants. Open variation points allow the inclusion of new variants to an existing variation points at runtime. Note than adding a variant to an existing variation point is quite different from adding a new variation point, but this concept seems really hard to implement. Therefore, the evolution of the variation points has impact in the evolution of the products.

2.2 Binding Time and Implementation Mechanisms

The selection of a particular feature implies the selection of its corresponding variation point, variants, and values. The realization of the variability, that is, the moment in which the variability happens, is often called the binding time (i.e.: *variability in time*). This binding time may take place at different moments, like design time, compilation, programming, run-time, etc. In [11], the authors identify different binding times for which variability may occur. Complementary to the definition of each particular binding time, the variation points defined at the design level must be implemented at the code level. Some of the alternatives to realize the variability are the following.

- **Compiler directives and installation procedures**: The variation points are realized when the software is compiled and this is achieved through directives, (e.g.: `#ifdef`). These directives can be used during the installation of software packages (e.g.: operating systems) to install or configure a particular product.
- **Flow control sentences**: Flow control sentences (e.g.: `if..then)` can be used at programming and run-time for selecting concrete options in the product.
- **Parameterization**: Variation points and variants are specified using function parameters that are instantiated at a given moment.
- **Boolean formula**: The result of a variation point is computed by means of a specific formula, usually based on logical connectors, and checked afterwards to select a particular alternative or a different variation point. Variability is implemented at programming time and realized at run-time.
- **Configuration**: Configuration files are loaded at the beginning of the execution of the system and used to select between different options. This configuration file is usually written in the same language of the software product.
- **Generator files**: Under a generative approach, makefiles can be used to deploy automatically a software package or particular product configuration, which is usually built on the top of different base software packages.
- **Database**: This mechanism is sometimes used (e.g.: context-aware systems) and the variations are realized depending of the values stored in databases. An initial set-up of the values must be done prior to the execution of the system, so they can be checked afterwards to decide between different alternatives.

Other additional techniques are possible to enable variability in general (e.g.: aggregation, inheritance, overloading, macros), such as described in [23]. In many cases and due to the complexity of software systems, several variability implementation mechanisms might be combined to obtain the desired configuration. In [29], the authors present a taxonomy of variability realization techniques in which different ways to implement the variation points are described. The authors describe the motivation of each realization technique and they relate it with a concrete stage of the lifecycle as well as the proposed solution for each case.

2.3 Dependency Model and Traceability

An important characteristic that affects both the modeling and the implementation of the variability is the existence of dependencies that can be established between features. A dependency may be originated because some feature needs the existence of another or because the modification of a particular feature impacts on other features. System constraints can be modeled as dependencies which limits the number of allowed products. During the modeling process, new dependencies add a degree of complexity to the features model with a direct impact on the definition and selection of the variation points. Feature models can use logical connectors like AND, OR and XOR to model the relationships between features, and variation points can be defined using these logical connectors. More complex dependencies can be modeled in a different way. Jaring and Bosch [15] discuss a taxonomy of variability realization techniques as a major concern when modeling and configuring products in a product line context. The authors identify four main types of variability dependencies, each of them consisting of four subtypes. Lee and Kang [20] suggest a classification of dependency types for feature modeling in product lines and they analyze feature dependencies that can be useful in the design of reusable components. Because feature modeling mainly focuses on structural dependencies, the authors propose to extend classic feature models by adding operational dependencies. An operational dependency is a relationship between features during the execution of the system. For instance, the *usage* dependency happens when a feature may depend on other features for its correct functioning. In [21], the authors describe a feature dependency model for managing static and dynamic dependencies for product line development. Three static dependencies and seven dynamic ones are defined. A directed graph is used to analyze domain requirements dependencies to produce the right product members in the product line. An algorithm generates the maximum connective dependencies graph but only direct dependencies are represented, no implicit ones. In general, variability modeling and management techniques are not enough powerful to support traceability, making it necessary to use other mechanisms to relate, for example, feature models with products. As stated in [7], capturing and representing this traceability is a challenging task. The authors propose a unified approach for successful variability management, in which trace links are defined to connect both the problem space (the feature model) with the solution space (instantiated architecture and code). Hence, several dimensions of variability can be defined to represent the variation points, the dependencies and the traces under a product line context. For the representation of these traces, and providing basic dependency types are handled, matrixes are a suitable option.

2.4 Variability Management

Variability is the ability to change or customize a system [32], and variability management includes the processes for maintaining the variability model across the different stages of the lifecycle in order to produce the right product configurations. Because features and variation points are not isolated, changes performed over a variation point may affect other variation points as well as the final product. Managing the variation points, variants and dependencies of a software system constitutes a big challenge for current variability modeling tools. Typical management activities should include: maintain the variation points, variants and dependencies, constraint checking, traceability between the model and products, configuration processes, and documentation. In [32], the authors describe some activities concerning variability management in a product line context, such as: variability identification, introduction of the variability in the system, collecting the variants and binding the system to one variant. The authors state the existence of feature interaction but the maintenance of the dependencies between features should be described explicitly in the proposed tasks. The technical report described in [22] discusses family based development processes and how to express requirements in terms of features and features in terms of variation points. Variability should be managed not only in the problem space, but also in the solution space during the development phase because the product portfolio has a direct impact on variability management activities.

3. Variability Representation and Visualization Techniques

The notation of variation point was introduced in RSEB (Reuse-Driven Software Engineering Business) [13], but some other notations have been proposed and used. In [23], the author analyses and compares five different leading modeling notations for representing variability in software systems (i.e.: FODA, FORM, Generative programming, Feature-RSEB and Jan Bosch's notation). Some of the aforementioned notations share FODA and FORM feature types and relationships while others propose extensions to the feature modeling. For instance, Jan Bosch's model proposes the introduction of "external features" which does not fit in the usual classification. A feature tree is one of the most common presentation techniques used for describing the variability of a system. In addition, UML uses some extensions to describe the variability of systems, like: UML stereotypes, tagged values and constraints based on the OCL (Object Constraint Language) [10]. In general, standard UML suffers the lack of a more precise notation to express all the variability concepts needed, the same as the original FODA feature trees which are not enough powerful to represent complex relationships between the variation points. Practical usage, however, suggests that FODA feature trees are well adapted for variability analysis, while the usage of UML profiles for variability has a clear focus on architecture, as described in [4].

3.1 Visualization

Variability management and modeling tools usually provide a graphical description of the variation points and variants for each product. Therefore, visualization and configuration facilities must be included, such as the following ones.

- **Tree view**: FODA trees are widely used for representing the variation points and variants for a particular product. One of the problems with this approach is the scalability when the number of alternatives grows.
- **Graph view**: Graphs are similar to trees but some kind of mechanism to expand the branches containing the alternatives is needed.
- **Matrix view**: Matrixes allow the visualization of a large number of items, but using this approach we can loose the perspective of the hierarchy and dependencies of the variation points and variants.

In addition, combo and check boxes or radio buttons can be used for the selection of the choices during product configuration. As the number of variation points and variants becomes unmanageable, the visualization of all the alternatives becomes a big problem. A way to solve this is to split the variability model into categories for which the user may select or visualize a portion of the tree. Another solution is to employ zooming tools to expand only those parts of the model in which the designer is interested on. The visualization of hundreds of variations points becomes an important problem in industrial product lines because it may hamper the scalability of visualization facilities.

4. State of Practice of Variability Modeling Tools

Once we have described the main concepts of software variability modeling, in this section we outline the main characteristics of some existing variability modeling and management tools. The tools selected offer specific functionality for modeling variability. The assessment was performed, based on the information given by the tools' authors or providers, who were asked to complete a review form. There was no cross-comparison between tools since the main objective of the evaluation is not to identify the best tool, but to identify the most relevant items that offer practical relevance to the community. The scope of this analysis is mainly focused on variability modeling and management tools rather than those using MDA-MDD [34], domain-specific languages [30], and generative approaches.

4.1 GEARS

Gears is a commercial software product line development tool developed by BigLever Inc [19] (http://www.biglever.com) and enables the modeling of optional and varying features which is used to differentiate the products in the portfolio. The Gears feature model uses rich typing (sets, enumerations, records, Boolean, integer, float, character, string) distinguishes between “features” at the domain modeling level and “variation points” at the implementation level (source code, requirements, test cases, documentation). In Gears, set types allow the selection of optional subsets, enumeration types allow selection of one and only one alternative, Boolean represent singular options, and records represent mandatory lists of features. Gears variation points are inserted to support implementation level variation. Components with Gears variation points become reusable core assets that are automatically composed and configured into product instances. Thus developers work in a very conventional way on Gears core assets, with the exception of implementing the variation points to support the required feature model variations that are in the scope of their asset.

Dependencies in Gears are expressed as relational assertions. Simple binary relations can be used to express the conventional require and excludes dependencies. Assertions can also contain 3 or more feature and relations such as “greater than”. Variation points and feature models are fully user programmable to arbitrary levels of sophistication and complexity. User-defined compound features have anonymous types (i.e., with the advanced typing in Gears, explicit user defined types and reusable types are not required).

The Gears approach defines product feature profiles for each product and selects the desired choices in the feature model. A product configurator automatically produces the individual products in the portfolio by assembling the assets and customizing the variation points within those assets to produce a particular product according to the feature profile. Gears modules can be mapped to any existing modularity capabilities in software. Gears modules can be composed into subsystems, which can be treated as standalone “product lines”. Product lines can be composed from modules and other nested product lines. Aspect-oriented features are captured in Gears “mixins”, which allow crosscutting features to be imported into one or more modules for use in implementation variation points in those modules. The tool supports also the definition of hierarchical product lines by nesting one product line into another.

Two views and editor styles are supported and can be switched dynamically: (1) syntactically and semantically well-defined text view and (2) context-sensitive structural tree view. Gears uses file and text based composition and configuration. This language-independent approach allows users to transition legacy variation as well as implement new variations. Gears has been used for all of the above and supports multiple binding times in one product line. For runtime binding, Gears typically influences the runtime behavior indirectly through statically instantiated configuration files or database setting, though these could also be set dynamically by making feature selections at runtime.

Gears technology eases the software mass customization process because enables organizations to quickly adopt a software mass customization for product line development. Gears supports three different models for product line adoption. Proactive, reactive and extractive approaches can be used depending of each particular organization, but they are not necessarily mutually exclusive. Gears has been used in systems with millions of LoC with no perceived limitation on scalability.

4.2 V-Manage

V-Manage from European Software Institute (ESI) (http://www.esi.es), is a tool for internal use that supports system family engineering in the context of MDA (Model Driven Architecture) and consists of the following three modules [26].

- **V-Define**: Defines the variation model (i.e. decision model in V-Manage terminology) as well as the relationships. The elements of the feature model are HTML links.
- **V-Resolve**: Builds application models by setting the values of the decision model and produces a suitable configuration of the decision model. It supports the resolution of the model using the variation model.
- **V-Implement**: Supports the implementation of reusable components and links the variation parameters attached to the decisions to some external components or to other dependencies. V-Manage generates the result of a particular configuration to HTML, PLC-code, a UML model or requirements document.

As described in [4], with V-Manage the user specifies a variation model and a resolution model, and provides a mechanism to specify product line models to derive concrete system models. The V-define interface is used as the front-end to specify the variation model for a product line. The resulting model is an application model where all the variations have been resolved. Dependency rules guide the user during the configuration of the values for each variation element. Binding and refinement are supported by V-implement for the production of reusable components.

4.3 COVAMOF

The COVAMOF (ConIPF Variability Modeling Framework, http://www.covamof.com) approach is a variability modeling approach for representing variation points and variants on all abstractions layers, supports the modeling of relations between dependencies, provides traceability, and a hierarchical organization of variability. Five types of variation points supported in COVAMOF: optional, alternative, optional-variant, variant and value. The optional-variant variation point refers to the selection (zero or more) from the one or more associated variants. The COVAMOF Variability View (CVV) [25] represents the view of the variability for the product family artifacts and unifies the variability on all layers of abstraction. The CVV models the dependencies that occur in industrial product families to restrict the binding of one or more variation points. Simple dependencies are expressed by a Boolean expression, and CVV specifies a function valid to indicate whether a dependency is violated or not. In addition to the Booleans, dependencies and constraints can also contain integer values, with operators ADD, SUBSTRACT, etc. Boolean and Numerical are used together in operators like the GREATER THAN, where numerical values are the input and Booleans are the output. Complex dependencies are defined in COVAMOF as dynamically analyzable dependencies and the CVV contains for each dynamically analyzable dependency the following properties [27]:

- **Aspect**: Each dependency is associated to an aspect that can be expressed by a real value.
- **Valid range**: This dependency specifies a function to {true, false} indicating whether a value is acceptable.
- **Associations**: The CVV distinguishes three types of associations for dynamic dependencies, which are: predictable, directional and unknown.

COVAMOF provides a graphical representation and a XML representation, used for communication between tools. The Mocca tool has been also developed to manage the COVAMOF Variability View, and allows for multiples views of CVV. Mocca supports the management of the CVV from the variation point view and the dependency view [26]. Mocca is implemented in Java as extension to the Eclipse 3.0 platform. Some recent improvements to COVAMOF, in particular to the derivation process, are supported by COVAMOF-VS tool suite [28], which is a set of add-ins for Microsoft Visual Studio.NET. The COVAMOF-VS provides two main graphical views, that is the variation point view and the dependency view, as a way to maintain an integrated variability model. Finally, specific plug-ins can be added for supporting different variability implementation mechanisms.

4.4 VMWT

VMWT (Variability Modeling Web Tool) is a research prototype developed at the University Rey Juan Carlos of Madrid. This first prototype (http://triana.escet.urjc.es/VMWT/) is a web-based tool built with PHP and Ajax and running over Apache 2.0. VMWT stores and manages variation points and variants following a product line approach and enables to create product line projects for which a set of reusable existing assets can be associated. Before configuring a particular product, the variants that will be part of the variation points of the feature model must be added. Each variant can be associated to a particular code component and we can specify numeric values (quantitative values), ranges of values or a enumerated list can be specified. Once all the variants have been added, the variation points will be added to the code components. VMWT supports dependency rules and constraints for the variation points and variants already defined. The following Boolean relationships are allowed: AND, OR, XOR and NONE. In addition, more complex dependencies can be defined, such as requires and excludes. The tool allows constraint and dependency checking and we it computes the number of allowed configurations. This is quite useful when it is needed to estimate the cost of the products to be engineered. Finally, a FODA tree is visualized for selecting the options for each product and the selected configuration is then displayed to the user. The variation points and variants selected are included in a file attached to each code component. Documentation of the product line can be automatically generated as PDF documents.

4.5 AHEAD

The AHEAD (Algebraic Hierarchical Equations for Application Development) Tool Suite (AHEAD TS) (http://www.onekin.org) was developed to support the development of product lines using compositional programming techniques [3]. AHEAD TS has been used in distinct domains (i) to produce applications where features and variations are used in the production process [9] (ii) to produce a product line of portlets. The production process in software product lines require the use of features that have to be modeled as first-class entities. AHEAD distinguishes between "product features" and "built-in features". The former characterizes the product as such. The latter refers to variations on the associated process. The production processes are specified in using *Ant*, a popular scripting language from the Java community. AHEAD uses a step-wise refinement process based on the GenVoca methodology for incrementally adding features to the products belonging to a system family. The refinements supported by AHEAD are packaged in layers. The base layer contains the base artifacts and the lower layers allow the refinements needed to enhance the base artifacts with specific features. The AHEAD production process distinguishes between two different stages. The intra-layer production process specifies the tasks for producing a set of artifacts within a layer or upper layers. The inter-layer production process defined how layers should be intertwined to obtain the final product. An extension to AHEAD is described in [31], and a tool called XAK was developed for composing base and refinement artifacts in XML format. ATS was refactored into features to allow the integration with XAK The feature refactoring approach used in XAK decomposes legacy applications into a set of feature modules which can be added to a product line. AHEAD doesn't require manual intervention during the derivation process.

6. Discussion and Comparison

Complementary to the tools described before, we can find other approaches for feature and variability modeling. In [33], the authors present the Koala approach for modeling variability in software product families. Koala specifies the variability in the design by selecting the components and appropriate parameters. Another approach presented in [5] outlines the variability of a product family into two levels: the specification level and the realization level. The variability model defines the variation points and where these are implemented in the asset base. Static and dynamic variation points are allowed, whereas dependencies are modeled 1-to-1. In [12], the authors mention the Product-Line UML based Software Engineering Environment (PLUSEE), which is a tool for addressing multiple views of a software product line and check the consistency among these views. Variability is defined using the UML notation through different views (e.g.: case model, static model, collaboration model, statechart model and feature model) and each view is related to a particular stage in the software lifecycle. There are two versions of PLUSEE, one uses Rational Rose and the other Rational Rose RT. PLUSEE is able to produce a consistent multiple-view model and an executable model using Rose RT.

The FeaturePlugin mentioned in [1] is an Eclipse plug-in for feature modeling.

This tool follows the FODA approach and supports cardinality-based feature modeling, specialization of feature diagrams and configuration based on feature diagrams. The FeaturePlugin tool organizes features in trees and has some extra characteristics: ColorTypes, Depth and DisplayTypes which, for instance, are hidden in the V-Manage tool. The tree organization of FeaturePlugin supports very easily the scalability of the tool when new characteristics are added. The BVR model [24] (developed under the European FAMILIES project) defines a meta-model for modeling the variability in system families. This meta-model has three main parts. The Base model is any model in a given language. The Variation model which contains variation elements referred to the Base model element. The Resolution model resolves the variability for a system family model. The BVR approach uses a prototype tool called Object-Oriented Feature Modeler (OOFM) made as an Eclipse plugin for supporting the feature modeling process. The ASADAL (A System Analysis and Design Aid Tool) described in [18] supports the entire lifecycle of software development process of a software product line and based on the FORM method [17]. ASADAL is a more complete approach compared to variability management tools like COVAMOF but variability management is a key feature of ASADAL. Two feature analysis editors for feature modeling and feature binding are implemented in ASADAL, and product-specific design can be instantiated through feature selection. ASADAL is able to generate executable code from an architecture model.

Table 1 describes a comparative analysis of the representative tools introduced in Section 4, for modeling and managing variability. As evaluation items, we have chosen the specific concepts about variability modeling and management most widely accepted (which were defined in Section 2). The characteristics described in the table were obtained analyzing the information available for each tool. We couldn't perform a real evaluation of some of the tools because GEARS is a commercial tool and no demo is yet available whereas V-Manage is for internal use. The VMWT was tested in our university and we obtained some of the characteristics from the remainder tools by interviewing the authors.

Table. 1. Comparative analysis of variability modeling tools

	GEARS	V-Manage	COVAMOF	VMWT	AHEAD
VP / Feature dependencies	Binary relations	AND, OR, NONE	Boolean Numerical	AND, OR, XOR, NONE	AND, OR, XOR, NONE
Complex dependencies	Relational assertions. Assertions can contain 3 or more feature and relations such as >, >=, <, <=, ==, subset, superset, AND, OR, NOT, +, -, *, / Require Excludes	----	Dynamic dependencies {aspect, valid range, associations}	Requires Excludes	Require Excludes
Feature / VP types	Set types allow selection of optional subsets. Enumeration types allow selection of one	Mandatory Alternative Optional Enabled Disabled	Alternative Optional Optional-variant Variant Value	Mandatory Alternative Optional	Mandatory Alternative Optional

	and only one alternative. Boolean types represent singular options Records are mandatory lists of features				
Traceability	Yes	Yes	Yes	No	Yes
Feature-VP allowed values	Integers Floats Characters Strings Booleans Atoms Ranges are constrained by assertions	XML like data values	Ranges of values	Numerical Enumeration Boolean String Ranges of values	-----
VP visualization	Feature tree Text view	Feature tree Feature list Flowchart to be planned	Feature tree	Feature tree	Feature tree
Extensibility Opened / Closed VP	Open for dynamic loading & component swapping Closed	Closed	Open & Closed (e.g: Open in SOA projects)	Closed	Closed
Variability Management facilities	5	4 (working on visual modeling)	5	4	4
Variability implementation mechanism	Composition Configuration	Parameterization Configuration Generation Macros Architectural design patterns Dynamic link libraries Dynamic class loading	Configuration Generation Macros Dynamic link libraries Parameterization	Parameterization Configuration	Inheritance Overloading
Integration of VP with software components	5	4 (working on plug-ins architecture and eclipse integration)	5 (VP are specified in or even extracted from the source code)	3	5
Scalability for visualizing VP	5	5	5 (Plug-in for additional views. visualizes the VPs in Visio)	3	4
Scalability	5	5	5	4	5
Feature dependency / constraint checking	Yes	Yes	Yes	Yes	Yes (SAT solver)
Automatic VP generation	Yes	Yes	Yes	Yes	Yes
Binding Time	Pre-compilation Compilation Linking Installation Startup	Pre-Compilation	Pre-compilation Compilation Linking Installation Startup	Programming Runtime	Pre- Compilation

	Runtime		Runtime		
Statistical analysis and Reporting	Yes	Yes	Yes	Variability and Project PDF documents	No
Product derivation	Yes	Yes	Yes	Partially	Yes
Estimation of the number of products	Yes (Combinatory reporting of estimated instances)	Yes	No	Yes	Yes (SAT solver)
Phases of the lifecycle covered	Analysis Design Implementation	Requirements Design Implementation	Analysis Design Implementation	Design Implementation	Design Implementation
Tool approach	Modeling Management	Modeling	Modeling Management	Configuration Modeling Management	Configuration Modeling
Development approach	Specialization Compositional	Compositional Derivation	Compositional Generative	Specialization Compositional	Specialization Compositional Generative
Platform / Technology	Java Standalone version with Eclipse and Visual Studio	Java Ant XML Eclipse	MS Visual Studio.NET XML	PHP AJAX JavaScript	Java-Ant XML

5. Impact on Product Line Development

Today, we have many examples of successful products lines (e.g.: Celsius Tech Cummins, Salion, Market Maker, Thales Naval, HP, etc) in which multiple products are designed and built under strong time to market conditions. The Nokia product line an example of successful product line which comprises multiple mobile phones organized around different families (e.g.: Nokia series 30, 60). A few examples of using the tools in real cases have been documented and the numbers of savings in terms of cost and effort have been reported. Some of the aforementioned tools have been tested both in academia and in industry. For instance, COVAMOF was used on the Intrada product family from Dacolian B.V., a small independent SME in The Netherlands for intelligent traffic systems [28]. Engenio is a firm dedicated to high-performance disk storage systems with approximately 200 developers distributed across four locations. Around 80% of the code is common to the 82 products of the firm. The increasing demand for Engenio's RAID storage server products led to the adoption of a product line approach and Gears was selected for this. Several successful results were obtained (see [14]). For instance, Engenio has experienced a 50% increase in development capacity. All these stories prove that software product lines constitute a successful approach for building software system families, and tools are strongly needed for managing the amount of variability required.

6. Current Limits and Second-generation Tools

The aforementioned tools constitute an important help for product line engineering. The lack of a unified approach for software variability leads to a certain number of tools with the same goal and similar characteristics, but with differences between them. There are some limitations of current tools as well as some issues that can be enhanced. For instance, the visualization of hundreds of variation points with their associated variants, in particular in industrial product lines, is a limitation of some of the existing tools, and new visualization facilities are welcome. As an example, we tested VMWT in two medium-size web projects and we observed that the tool needs better visualization capabilities when the number of variation points and variants scale up. Another issue, in particular for complex systems, is the need to handle complex dependencies. Some of the existing tools need to support more complex dependencies for describing all the relationships and constraints for any type of software system. Also, the extensibility of the variability model for supporting runtime variability (e.g.: adding variation points during the execution of the system) is a complex problem associated to the evolution of the system that could be alleviated by the usage of plug-ins mechanisms in runtime. The integration of variability modeling tools with traditional software engineering tools seems quite interesting for software product line engineering. In particular, unless there is a serious effort by the variability tools providers to integrate these tools with software configuration management tools and integrate the variability management activities in the practical development processes, the success of these tools will be in danger. The integration of variability tools with the configuration management community is an interesting issue to explore for configuration management purposes. Also, variability management tools could learn about the knowledge management community to incorporate more additional features. These and other capabilities seem to be interesting to be added and we believe a second-generation variability tools is a goal to pursuit. In this paper we have studied some representative tools for modeling and managing software variability in order to discover the needs for second-generation tools. Most of the tools analyzed share many common characteristics and most of them focus on specialization and compositional approaches. The estimation of the number of allowed products or right configurations is an interesting issue for the defining production plans and for cost estimation. The majority of the tools cover design and implementation phases of the software lifecycle. Finally, integrated derivation processes from feature models to products are welcome to reduce the effort of configuration processes.

Acknowledgements

We thank to Charlie Krueger for his useful information about GEARS, which has been balanced between commercial and academic reasons; also Jason X. Mansell (ESI) who gave us many responses about V-manage. Many thanks to Marco Sinnema for providing useful information of COVAMOF-VS, and Salva Trujillo and Oscar Díaz who provide us the details of AHEAD TS in alive discussions

References

1. Antkiewicz, M. and Czarnecki K. FeaturePlugin: Feature Modeling Plug-in for Eclipse, OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, 2004
2. Bass, L., Clements P. and Kazman, R. Software Architecture in Practice, Addison-Wesley, 2nd edition, (2003).
3. Batory, D., Sarvela, J.N. and Rauschmayer, A. Scaling Step-Wise Refinement. IEEE TSE 30(6) 355-371, (2004).
4. Bayer, J, Gerard, S., Haugen, Ø, Mansell, J., Møller-Pedersen, B., Oldevik, J., Tessier, P., Thibault, J.P. and Widen, T. Consolidated Product Line Variability Modeling, In Software Product Lines Research Issues in Engineering and Management, Springer-Verlag, Timo Käköla and Juan Carlos Dueñas (Eds), pp. (2006).
5. Becker, M. Mapping Variability's onto Product Family Assets. Procs of the International Colloquium of the Sonderforschungberich 501, University of Kaisersalutern, Germany, (2003).
6. Bosch, J. Design and Use of Software Architectures, Addison-Wesley (2000).
7. Berg, K., Bishop, J. and Muthig, D. Tracing Software Product Line Variability – From Problem to Solution Space. Procs of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT), pp. 111-120 (2005).
8. Capilla, R. and Dueñas, J.C. Modelling Variability with Features in Distributed Architectures, Procs of Product Family Engineering (PFE), Springer-Verlag LNCS pp. 319-329, (2001).
9. Díaz, O., Trujillo, S. and Anfurrutia, F.I. Supporting Production Strategies as Refinements of the Production Process. Procs of 9th Software Product Line Conference (SPLC), Springer-Verlag LNCS 3714 pp.210-221, (2005).
10. Dobrica, L. and Niemelä, E. Using UML Notation Extensions to Model Variability in Product-line Architectures. International Workshop on Software Variability Management (SVM), ICSE'03, Portland, Oregon, USA pp. 8-13 (2003).
11. Fritsch, C., Lehn, A. and Strohm, T. Evaluating Variability Implementation Mechanisms. Procs of International Workshop on Product Line Engineering (PLEES'02), Technical Report at Fraunhofer IESE (No. 056.02/E) 59-64 (2002).
12. Gomaa, H. and Shin, M.E. Variability in Multiple-View Models of Software Product Lines. International Workshop on Software Variability Management (SVM), ICSE'03, Portland, Oregon, USA pp. 63-68 (2003).
13. Griss M. L., Favaro J., d'Alessandro M., Integrating Features Modeling with the RSEB. 5th International Conference on Software Reuse, IEEE Computer Society (1998)
14. Hetrick, W.A., Krueger, C.W. and Moore, J.G. Incremental Return on Incremental Investment: Engenio's Transition to Software Product Line Practice. Conference on Object Oriented Programming Systems Languages and Applications, Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA), ACM pp. 798-804 (2006).
15. Jaring, M. and Bosch, J. Variability Dependencies in Product Family Engineering. 5th International Workshop on Product family Engineering (PFE), Springer-Verlag, LNCS 3014, pp. 81-97, (2004).
16. Kang K. C., Cohen S., Hess J. A., Novak W. E., Peterson A. S.. Featured-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21 ESD-90-TR-22, Software Engineering Institute, Carnegie Mellon University, Pittsburgh (1990).
17. Kang K. C., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. FORM: A Feature-oriented Reuse Method with Domain Specific Software Architectures. Annals of Software Engineering, Vol 5(1) pp. 143-168, Springer (1998).

18. Kim, K., Kim, H., Ahn, M., Seo, M., Chang, Y. and Kang, K.C. ASADAL: A Tool System for Co-Development of Software and Test Environment based on Product Line Engineering. ICSE 2006, pp. 783-786, (2006).
19. Krueger, C. W. Software Mass Customization. BigLever Software Inc. Available at: http://www.biglever.com/extras/BigLeverMassCustomization.pdf, (2006).
20. Lee, K. and Kang, K.C. Feature Dependency Analysis for Product Line Component Design. 8th International Conference on Software Reuse (ICSR), Madrid, Springer-Verlag LNCS 3107, pp. 69-85, (2004).
21. Lee, Y., Yang, C., Zhu, C. and Zhao, W. An Approach to Managing Feature Dependencies for Product releasing in Software Product Lines. 9th International Conference on Software Reuse (ICSR), Turin, Italy, Springer-Verlag LNCS 4039, pp. 127-141, (2006).
22. Myllymäki, T. Variability Management in Software Product Lines. Technical Report 30. Institute of Software Systems, Tampere University of Technology (2002).
23. Robak, S. Feature Modeling Notations for System Families. International Workshop on Software Variability Management (SVM), ICSE'03, Portland, Oregon, USA pp. 58-62 (2003).
24. Shakari, P. and Møller-Pedersen, B. On the Implementation of a Tool for feature Modeling with a base Model Twist. Available at: http://www.himolde.no/nik06/articles/08-Shakari.pdf
25. Sinnema, M., Deelstra, S., Nijuis, J. And Bosch, J. COVAMOF: A Framework for Modeling Variability in Software Product Families. Procs of 3rd International Software Product Line Conference (SPLC), Springer-Verlag LNCS 3154, pp. 197-213, (2004).
26. Sinnema, M., de Graaf, O. and Bosch, J. Tool Support for COVAMOF. Procs of the 2nd Groningen Workshop on Software Variability Management (SVMG), Groningen, The Netherlands, (2004).
27. Sinnema, M., Deelstra, S., Nijuis, J. And Bosch, J. Modeling Dependencies in Product Families with COVAMOF. Procs of 13th International Workshop on Engineering of Computer Based Systems (ECBS'06), pp. 299-307 (2006).
28. Sinnema, M., Deelstra, S., Nijuis, J. and Hoekstra, P. The COVAMOF Derivation Process, 9th International Conference on Software Reuse (ICSR), Turin, Italy, Springer-Verlag LNCS 4039, pp. 101-114, (2006).
29. Svahnberg, M., van Gurp, J. and Bosch, J. A Taxonomy of Variability Realization Techniques. Software Practice & Experience, vol 35(8), 705-754, (2005).
30. Tolvanen, J-P., Kelly, S. Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences, Proceedings of the 9th International Software Product Line Conference, H. Obbink and K. Pohl (Eds.) Springer-Verlag, LNCS 3714, pp. 198 – 209, (2005).
31. Trujillo, S., Batory, D. and Díaz O. Feature Refactoring a Multi-Representation Application into a Product Line. Procs of 5th International Conference on Generative Programming and Component Engineering, pp. 191-200 (2006).
32. van Gurp, J., Bosch, J. and Svahnberg, M. Managing Variability in Software Product Lines. Procs of IEEE/IFIP Conference on Software Architecture, WICSA 2001, Amsterdam, The Netherlands, IEEE CS, pp. 45-54 (2001).
33. van Ommering, R. van der Linden, F., Kramer, J. and Magee, J. The Koala Component Model for Consumer Electronics Software. IEEE Computer, pp. 78-85, (2000).
34. Oldevik, J. Solberg, A., Haugen, Ø., -Pedersen, B. Evaluation framework for Model-Driven Product Line Engineering tools, In Software Product Lines Research Issues in Engineering and Management, Springer-Verlag, Timo Käköla and Juan Carlos Dueñas (Eds), pp. (2006).

Tool-Supported Multi-Level Language Evolution

Markus Pizka and Elmar Jürgens

Technische Universität München
Institut für Informatik
Boltzmannstr. 3
Germany – 85748 Garching

Abstract. Through their high degree of specialization, domain specific languages (DSLs) promise higher productivity and thus shorter development time and lower costs than general purpose programming languages. Since many domains are subject to continuous evolution, the associated DSLs inevitably have to evolve too, to retain their value. However, the continuous evolution of a DSL itself can be very expensive, since its compiler as well as existing words (i. e. programs) have to be adapted according to the changes to a DSL's specification. These maintenance costs compromise the expected reduction of development costs and thus limit the success of domain specific languages in practice.
This paper proposes a concept and a tool for the evolutionary development of domain specific languages. It provides language evolution operations that automate the adaptation of the compiler and existing DSL programs according to changes to the DSL specification. This significantly reduces the cost of DSL maintenance and paves the ground for bottom-up development of domain specific languages.

1 Domain Specific Chances and Limitations

Albeit three decades of intense research and significant progress in research and practice, the development and maintenance of software systems still constitutes a time-consuming, costly and risky endeavor. The reduction of software development and maintenance costs thus remains a research topic of paramount importance to software engineering. A basic idea behind many approaches that attempt to reduce these costs is to increase the productivity of software developers. One approach to raise productivity that has received increased attention in recent years, are domain specific languages.

1.1 Productivity Through Specialization

A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [1]. Hence, the key characteristic of domain specific languages is their specialization to a problem domain. This specialization allows them to offer language constructs and abstractions tailored to the class of problems from this domain. DSLs typically allow these problems to be described very directly and concisely, requiring less developer effort than general purpose programming languages. DSLs thus have the potential to increase productivity and decrease costs of software development. Particularly prominent examples for the benefits of DSLs are found in the compiler construction field with specialized languages and generators (amongst others) for hashing (e. g. gperf), scanners, and parsers. Tools for the interactive design of graphical user interfaces, the query language SQL, interface definition languages like WSDL or modeling languages like MATLAB/Simulink add further examples for the benefits of DSLs, i. e. purpose-built languages with powerful generators.

Besides reduced development time and cost, DSLs are a promising mean to decrease the maintenance cost of software because of the reduced code size and increased comprehensibility, both due to the higher expressiveness of DSLs compared to general-purpose languages. The decrease of maintenance costs is particularly important since 60-80% of the costs of software are usually not devoted to initial development but to maintenance [2–4].

However, DSLs only help to reduce overall maintenance costs, as long as the costs for development and maintenance of the DSL itself can be amortized. If unanticipated changes to a domain require changes to the DSL definition, maintenance costs can be high, as noted in [5].

1.2 Limitations

Thus, the success of DSLs is clearly still limited to few niches. To the extent of our own practical experiences and that of our commercial partners, the bulk of code that gets written, be it information systems in financial institutions or flight control systems for commercial aircrafts, makes little use of DSLs. Languages like Risla for Financial Products [5] are rare exceptions to the rule.

We state that there are at least three major reasons for this which we will explain more precisely in the following paragraphs. First are the costs of designing and implementing DSLs. Second, the limited capabilities of one-step generation and third, the constant need to evolve DSLs.

The multi-level language evolution concept and tool-support presented in this paper largely increases the applicability of DSLs by helping to overcome exactly these three current core difficulties in DSL design and implementation.

2 Challenges in DSL Design and Maintenance

The goal of the concept and tool-set presented in this paper is to increase the long-term applicability of DSLs by supporting an evolutionary and bottom-up oriented style for DSL design and implementation. The rationale for this approach is the need to overcome three major obstacles that we detail in the following paragraphs.

2.1 DSLs Are Expensive to Build

Though DSLs promise substantial gains in productivity, the development of DSLs itself is expensive and troublesome. [6] summarizes the difficulties of building DSLs adequately as follows:

> DSL development is hard, requiring both domain knowledge and language development expertise. Few people have both. Not surprisingly, the decision to develop a DSL is often postponed indefinitely, if considered at all, and most DSLs never get beyond the application library stage.

Clearly, one of the primary contributions of DSLs is enabling reuse, i. e. reuse of abstractions and reuse of the knowledge about how to implement these abstraction in different contexts. As such, DSLs must cope with the same economic challenges like any other reuse oriented approach [7]. Building reusable components requires a costly analysis of the domain and its variability followed by an even more expensive implementation of reusable components[1]. The costs of planning and building such generalized components are usually a multiple of the costs of building a concrete solution to a particular problem [7]. Hence, building DSLs only pays-off after repeated successful use of the DSL. However, due to the constant change of requirements and the execution environment [8], the future use of a DSL is uncertain and building DSLs is economically risky. Note, that this risk increases with the degree of specialization respectively the potential benefit.

Requirement 1 ***(Stepwise Bottom-Up Generalization)***
To reduce the uncertainty of the benefit of DSL design and implementation, DSLs should be built in an incremental and bottom-up oriented manner instead of the currently predominant top-down and big-bang like approach. To support this style of DSL development, means for the step-wise generalization of existing concepts and solutions as needed are required.

If DSLs can be built by gradually abstracting and flexibilizing existing solutions (including DSLs) as needed, then the cost of building a DSL will never significantly exceed the costs of developing the desired new solution from scratch, but often provide immediate pay-offs. No effort has to be put into speculation about future requirements and there is no need for risky in advance investment into flexibility that could possibly be needed in the future.

2.2 Generators are no Oracles

A DSL usually requires a generator that reads a word[2] of the domain specific language and produces a word in the desired target language. Program generators for DSLs are nothing but program transformation systems providing a

[1] In case of DSLs represented through the domain language and a code generator.
[2] The term *word* is used as in formal language theory to denote strings that conform to the language syntax.

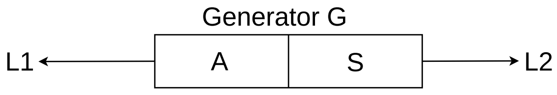


Fig. 1. Generator basics

translation from a higher to a lower level language, (also called *synthesis* or *compilation*). Hence, program generators are subject to the same inherent limitations as conventional compilers that translate from C to Assembler or Java to Byte-code, though program generators usually operate on a higher level of abstraction.

Figure 1 illustrates the basic structure of a program generator G. G reads words $v \in L_1$ and produces words $w \in L_2$ by first performing an analysis A of v and then synthesizing (S) result w. The whole benefit of this strategy corresponds to the distance between the level of abstractions of the input and the output languages, denoted by $A(L_1)$ and $A(L_2)$. Basically[3], there are three different possibilities:

$A(L_1) = A(L_2)$: The DSL L_1 and its generator G are useless from a productivity perspective. G does not contribute any decisions to the implementation. All details of the output w are already specified in the input v. G only rephrases v which might increase readability of v compared to w but not reduce its complexity.

$A(L_1) > A(L_2)$: This means that some details of w are not described in v but G decides on the implementation of these details. Examples are the allocation of memory for local variables in C compilers or the optimization of an SQL query. Here, the gains for the user of L_1 are obvious. By leaving some decisions on *how to* implement w up to G, v becomes shorter and more declarative, by describing G *what to* implement.

$A(L_1) >> A(L_2)$: Unfortunately, the possibility to stretch the distance between $A(L_1)$ and $A(L_2)$ is very limited because of computational complexity. Even basic decisions, like the allocation of registers, turn out to be NP-hard [9]. It is our conviction, that mapping higher level descriptions such as a financial service specification to Java classes and objects efficiently will have to cope with similar complexity issues. Usually this complexity is circumvented by accepting suboptimal decisions and thereby reduces quality of the output. The wider the gap between $A(L_1)$ and $A(L_2)$ gets, the less information will be available to the decision maker G resulting in a weaker result w in terms of performance, reliability, usability and so on. Clearly, reduced quality is counterproductive for reuse. McIlroy stated in 1968 *"No user of a particular member of a family should pay a penalty in unwanted generality"* [10]. E. g. current Object-Relational mapping tools suffer from this trade-off.

This leads to the following contradictory observation:

[3] Ignoring reverse engineering, since it cannot increase productivity, where $A(L_1) < A(L_2)$

1. The benefit that can be gained from a single generator is inherently and severely limited. We are convinced that there will not be a single flexible DSL for some high-level business domain with a generator that maps it to high-quality Java code.
2. To gain a significant advantage from a DSL, the gap between the level of abstractions of the input and the output has to be wide (see $A(L_1) > A(L_2)$).

The concept and tool-set presented in this paper aims at solving this concept by fulfilling the following requirement.

Requirement 2 ***(DSL Layering)***
The design and implementation of a DSL should not be limited to a one-step compilation but support layers of DSLs and a staged generation process with additional user input at each stage.

Note, that staging further increases the complexity and costs of building and maintaining the DSL as discussed in 2.1 because changes on one stage might affect other stages, too. Again, ruling this complexity requires a tool-set that aids in gradually adapting the DSL hierarchy as needed (see requirement 1).

2.3 Language and Word Evolution

While building a DSL is costly, building layered DSLs is even more expensive and maintaining a single or even layered DSL is even worse because nothing is more constant than change entailing a constant need for evolution [8].

The design and implementation of a DSL trivially depends on the requirements of the domain. With the exception of DSLs that model a technical domain, such as regular expressions or SQL, the requirements are directly connected with the business processes in this domain. Unfortunately, business processes are very volatile [11], simply because business process agility is the mean to achieve competitive advantages.

This poses a serious difficulty for DSLs. On the one hand, a DSL should be high-level or in other words as close to the business processes as possible in order to provide increased productivity compared to a general purpose language. On the other hand, the tighter the DSL is connected with the business processes, the more fragile it gets and the more often it will have to be changed, which in turn reduces the benefits of possible reuse.

A non-trivial change to an existing DSL L leading to a new DSL version L' requires the following three major steps:

1. Change of the definition of L – its syntax and semantics.
2. Adaption of all tools processing L; at least the corresponding compiler or generator but possibly also syntax aware editors (e. g. highlighting), debugger, etc.
3. Transformation of all already existing words (programs) $w \in L$ into language L'.

As an alternative to step 3, one could also maintain older versions of DSLs so that words in older versions of the language could still be used and changed independently of newer versions of the language. However, this would create a complicated configuration management problem and in addition to this, users of older versions of the language could not benefit from any advantages of newer versions and new tools. In practice, this drawbacks forces users to migrate their words to the new version.

Hence, most DSLs will have to evolve over time, including the tools that process these DSLs and words written in these languages. Without adequate tool-support, DSL evolution is a complex, time-consuming, and error-prone task that severely hampers the long-term success of a DSL.

Requirement 3 ***(Automated Co-Evolution)*** *DSL maintenance is inevitable for most realistic domains and requires adequate tool support. The transformation of existing words and the adaptation of language processing tools according to changes of the language has to be automated as far as possible.*

Note that this requirement complements requirement 1 because the tool-supported co-evolution of language, tools and words is a contribution to stepwise bottom-up generalization as formulated in requirement 1.

3 Related Work

The work presented in this paper combines DSLs [1] and generative programming [12] with elements of program transformation [13] and compiler construction, as well as software evolution [8]. Within this general context, the evolution concept has strong relations with Grammar Engineering and language evolution as described in the following paragraphs.

3.1 Grammar Engineering

In [14], Klint, Lämmel and Verhoef argue that although grammars and related formalisms play a pervasive role in software systems, their engineering is insufficiently understood. They propose an agenda that is meant to promote research on Grammarware and state research challenges that need to be addressed in order to improve the development of grammars and dependent software.

One of these challenges is the development of a framework for grammar transformations and the co-evolution of grammar-dependent software. The Grammar Evolution Language proposed in this paper offers such grammar transformation operations, and the automatic generation of compilers from DSL definitions with static validation of path expressions aims at the desired co-evolution of one important instance of grammar-dependent software, namely the compiler.

In [15], Lämmel proposes a comprehensive suite of grammar transformation operations for the incremental adaptation of context free grammars. The proposed operations are based on sound, formal preservation properties that allow to reason about the relationship between grammars before and after transformation. [16] and [17] present systems that implemented these evolution operations to incrementally transform LLL and SDF grammars.

Lämmel's grammar adaptation operations inspired the design of the Grammar Evolution Language used in our approach as a mean to automate language evolution. However, this paper focuses primarily on the coupled evolution of grammars and words of the language described by these grammars. Compared to the operations suggested by Lämmel, the Grammar Evolution Language sacrifices the formal basis to allow for simpler coupled evolution operations. It would be desirable to combine the coupled evolution capabilities proposed in this paper with the formal preservation properties proposed by Lämmel in future versions of our tool Lever.

3.2 Evolution of Language Specifications

TransformGen is a system that generates converters that adapt programs according to changes of the language specification [18, 19]. While TransformGen automatically produces converters for local[4] changes, non-local transformations must be specified manually. Furthermore, non-local transformations cannot be reused between recurring evolution operations.

TransformGen only targets the adaptation of words but does not take language processing tools into account. The tool Lever presented in this paper goes one step further by semi-automating the adaptation of compilers, too. Moreover, Lever supports reuse of coupled transformations.

3.3 Schema Evolution in Object Oriented Databases

Regarding a data base schema as a language and the information contained in a data base as the words of this language allows to relate schema evolution with program transformation. Co-evolution of language and words is of predominant importance to this field and studied in various works.

In [20], Banerjee proposes a methodology for the development of schema evolution frameworks for object oriented databases (OODB) that was used in the ORION OODB system. The methodology suggests invariants for consistent database schemas and evolution primitives for incremental changes to the database. The evolution primitives perform coupled updates of both the schema and the objects in the database. Similar schema invariants and update primitives were proposed in [21] for GemStone OODB. The DSL Dictionary invariants that we use in our approach were inspired by these ideas. In [22], Ferrandina describes the schema evolution approach used in the O_2 OODB. In contrast to the coupled evolution primitives of ORION, it performs schema and data updates separately. While a declarative language is used for schema updates, migration of objects to new schema versions is based on user-defined conversion functions. Through this separation, O_2 is able to support global evolution operations. The

[4] Local transformations are restricted to the boundary of a grammar production.

approach presented in this paper extends this idea by separating grammar and word evolution language without requiring user-defined functions in a general purpose language.

The SERF schema evolution framework [23] uses OQL for schema and data manipulations and transformation templates, in order to provide extensible, coupled evolution operations. The transformation templates combine the advantages of the above mentioned approaches, by allowing both local and global transformations to be specified using expressive, coupled evolution operations affecting both schema and data objects. These transformation templates stimulated the use of Jython procedures to form language evolution statements by coupling grammar and word evolution language statements in the approach presented here.

4 Language Evolution Concept

The core concepts of the proposed approach to construct multi-level DSLs (see 4.1) incrementally are grammar, word, and language evolution languages (4.3), and a generator architecture that is built around DSL histories (4.4).

4.1 Divide and Conquer

According to requirement 2 of section 2, layering DSLs is a key design principle for building powerful DSLs that map high level specifications to their implementation. Figure 2 illustrates the difference between one-step generation and the layering proposed in this paper.

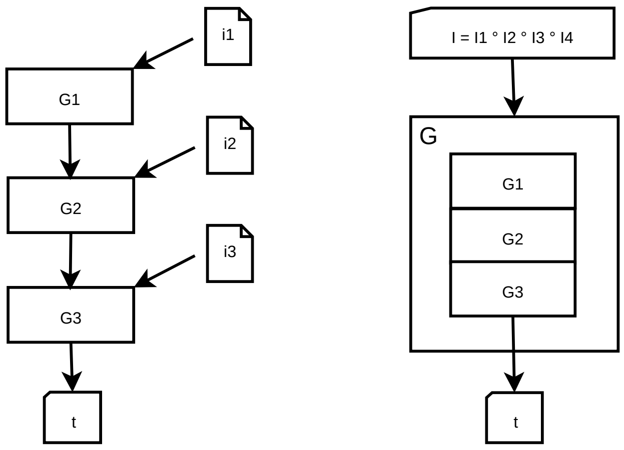


Fig. 2. Staged versus one-step generation

On a theoretical level, staging the compilation process as shown on the left into three generators G_1, G_2, and G_3 that produce the output t in a sequence, seems identical with one-step generation of a composed generator G as shown on the right hand side; with the technical exception that inputs i_1, i_2 and i_3 are not fed into the generation process at once but at the beginning of each stage. In fact, the concatenation of the various inputs $i = i_1 \circ i_2 \circ i_3$ could be regarded as a word of the language I that results from concatenating the input languages I_1, I_2, and I_3.

However, there are strong differences between the staged or the one-step generation model when it comes to the implementation of the DSL. Note, that not only the input fragments i_2 and i_3 depend on i_1 respectively $i_2 \circ i_1$ but also every language I_n depends on all inputs previous to stage n. Technically speaking, I_n corresponds with the information needed by G_n to further drive with the generation process in the situation created by $i_0, \ldots, i_{n_1}$. Now, specifying the unified language I of all possible input sequences would theoretically be possible but technically impractical. It would yield a undesirable DSL with numerous semantical conditions and exceptions allowing and restricting the use of language elements within a word of the language depending on arbitrary prefixes of the word.

In addition to the improved structuring of DSL, the staged model also indicates a feasible way of implementing complex generation process by dividing the task into separate steps with individual inputs at those points where it is needed. Though this might seem surprising for DSL design and implementation, this is exactly the strategy that system level software uses successfully to map high-level applications to system-level representations for execution. E. g. the C++ source code i_1 gets compiled with the C++ compiler G_1. The link-loader G_2 further sets the memory layout according to whether the user wants to execute the code as a stand-alone application or a shared library as specified in i_2. The operating system kernel G_3 then maps the results of these steps to main memory pages, CPU cycles, and so on according to the priorities of the user (i_3). Imagine the same process without staging. It would surely be possible but either hard to comprehend or less flexible.

4.2 Generator Maintenance By Transformation

Maintaining such a sequence of dependent generators is highly complex by itself and only practical with adequate tool-support. For example, if the top level DSL I_1 must be changed to accommodate a new feature, there is a high probability that the output of G_1 changes too, entailing the need for changing I_2, G_2, and so on. All of these changes of languages (I_n) and programs (G_n) can themselves be treated as language and program transformations. Hence, the evident tool to maintain a staged DSL is itself a DSL for the domain of DSL manipulation.

The crucial element of this overall architecture is the top-level language evolution language and the meta-level generator H. Our tool called Lever[5] presented in section 5 implements significant parts of such a meta-level generator based on grammar and word evolution languages.

[5] Language evolver.

4.3 Language Evolution Operations

The three proposed evolution languages for the manipulation of DSL specifications are displayed in Figure 3.

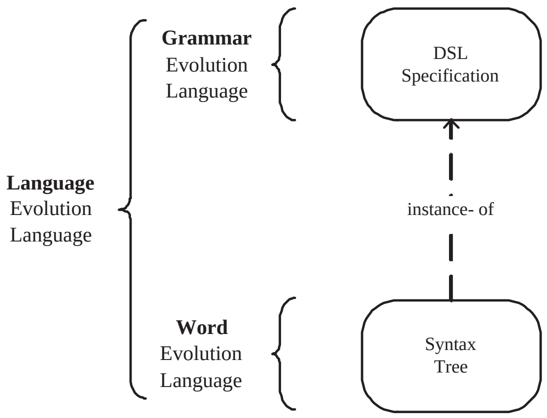

Grammar Evolution Language
DSL Specification
Language Evolution Language
instance- of
Word Evolution Language
Syntax Tree

Fig. 3. Evolution languages

Grammar Evolution Language (GEL) transforms the syntax and static and translational semantics of a DSL. GEL operations can be used for both creating the initial version as well as modifying it in order to yield subsequent versions of a DSL.
The GEL is complete in the sense that its statements can be used to transform any DSL syntax (and semantics) into any other DSL syntax (and semantics).

Word Evolution Language (WEL) statements work on the syntax trees of DSL words. During language evolution, they are used to perform syntax tree transformations to compensate changes of the underlying grammar.
WEL is complete in the sense that its statements can be used to transform any syntax tree into any other syntax tree and thus to compensate arbitrary changes to the DSL specification.

From the point of view of expressiveness, the combination of these two evolution languages allows the specification of all possible transformations that might arise during the evolutionary development of a DSL.

However, from the point of view of usability, a third evolution language is desirable: the grammar and word evolution languages merely provide a low level of abstraction. Even simple coupled evolution operations, such as renaming a keyword in the syntax and all existing words, require at least two evolution operations – one from each language. Furthermore, coupled transformation knowledge cannot be reused to simplify recurring evolution operations. This gap is filled by the third evolution language.

Language Evolution Language (LEL) statements perform coupled evolution of both the grammar and the syntax tree. They provide a higher level of abstraction to users and enable reuse of coupled transformation knowledge. LEL builds on the GEL and WEL to implement its transformations.

LEL can be conceived as a procedure mechanism that uses GEL and WEL statements in the bodies of LEL procedures.

4.4 Evolution Architecture

Figure 4 shows the central components of the architecture of our language evolution tool Lever.

All evolution operations applied during the construction and evolution of a DSL are stored in the DSL History. The DSL History thus contains transformation information that specifies the delta between consecutive versions of a DSL. This transformation information is used to automatically adapt both the DSL compiler and existing DSL words to conform to the latest language version.

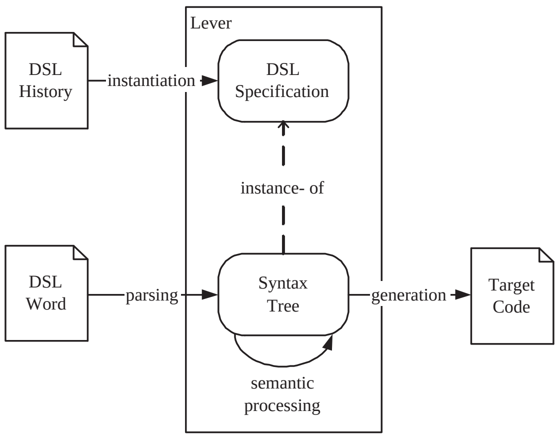


Fig. 4. Lever architecture

DSL History contains evolution operations that define specifications for all versions of a DSL: The first evolution operations create the DSL specification for the initial version of a DSL. Subsequent evolution operations transform the DSL specification to yield later DSL versions.

DSL Specification is a comprehensive, declarative specification of the syntax and static and translational semantics of a single version of the DSL. It is explicitly available at runtime and drives the compilation process.

Syntax Tree is the in-memory representation of DSL words. It is an abstract syntax tree that is decorated with concrete syntax and semantic attributes.

DSL Word is the input for the compilation process. DSL words are versioned to allow the identification of the DSL version in which the word was written.
Target Code is the result of the compilation process.

Compiling Words of Arbitrary Versions The information contained in the DSL history allows to translate DSL words written in any version of the DSL. During the compilation process, the following steps are performed:

1. Identification of the DSL word's language version.
2. Execution of the evolution operations from the DSL history in order to create a DSL specification in the corresponding language version.
3. Generation of a parser from the information in the DSL specification. The parser is then used to instantiate the syntax tree from the DSL word.
4. Transformation of DSL specification and syntax tree to the latest language version. Versions of the DSL dictionary and the syntax tree are compared with this latest DSL version. If needed, the DSL history is used to transform both the DSL specification and the syntax tree to the latest version.
5. Semantic processing: according to the DSL semantics contained in the DSL specification, target code for the syntax tree is computed and written to the output.

5 Implementation: Lever

The proposed evolution operations and evolution architecture is implemented prototypically in our tool Lever. Lever is implemented in Java. Evolution languages are implemented as internal DSLs in Jython [24], the *Scannerless Generalized LR* parser (SGLR) [25] is used for parsing and the *velocity template engine* is used for code generation.

5.1 DSL Specification Formalism

Lever uses an object oriented interpretation of attribute grammars[6] [26] as specification formalism for both syntax and semantics of a DSL. In Lever, DSL specifications are called *DSL Dictionaries*, since they define the syntax and semantics of every word a language comprises.

In DSL dictionaries, semantic rules specify how target code gets generated from the data contained in the syntax tree. In order to cleanly separate target code fragments, code generation logic and syntax tree access from one another, DSL dictionaries use code generation templates as semantic rules. Access to the syntax tree from within code generation templates runs via XPath [27] expressions.

Every access to the syntax tree from within a semantic rule introduces a dependency between the rule and the syntax tree. Language evolution operations

[6] Context free grammars extended with semantic attributes and rules for their computation.

may change the shape of the syntax tree and thus potentially break these dependencies. In order to support DSL architects, Lever can statically validate all XPath expressions against the DSL dictionary and thus detect broken dependencies during language evolution.

5.2 Evolution Operations in Lever

The Grammar Evolution Language (GEL) comprises statements to declare nonterminals, to create, rename and delete productions, to add, modify and remove (literal, terminal or nonterminal) production components and (inherited or synthesized) attribute declarations, to set semantic rules, to change the order of production components and to influence priorities and associativity of productions. Every GEL statement operates on a single DSL Dictionary element.

The Word Evolution Language (WEL) comprises statements that use XPath expressions to select, insert, update and remove nodes from the syntax tree. Furthermore, it contains statements to declaratively construct syntax tree fragments and change the dictionary element a syntax tree node instantiates.

The Language Evolution Language (LEL) comprises statements for recurring coupled evolution operations, such as the introduction or removal of literal or terminal symbols, the encapsulation or in-lining of production components or the renaming of productions or literals (i.e. keywords).

5.3 Limitations

The current version of Lever only automates the adaptation of the DSL compiler. Additional tools, such as a debugger, pretty printer or syntax aware editor still have to be maintained manually.

Furthermore, Lever currently only targets *textual* DSLs. However, it is our conviction, that the stated problems also hold for *visual* DSLs and we believe that the concepts this paper proposes can also be applied to them.

6 Case Study: Catalog Description Language

As a proof of concept, Lever was applied to develop a specification language for product catalog management systems in an evolutionary way. The results show the feasibility of the proposed approach to DSL development.

Due to space constraints, this case study only demonstrates language evolution in a single stage scenario. On a conceptual level, this can be justified, since the conceptual distance between the DSL and the target code framework is small enough to allow for generation of high-quality Java code.

6.1 Domain

Product catalogs are collections of structured product documents. Each document belongs to a product family. Typically, all documents within a product family share the same structure, whereas different families have different document structures.

Catalog management systems are used to create, manage and publish product catalogs. This comprises the creation, manipulation and deletion of documents by users, and the persistence and export of catalog data to different media. Catalog management systems are data-centric. Thus, solution domain artifacts that implement editors, display forms, persistence and data export depend on the structure of the documents the implemented catalog comprises. Implementing each artifact by hand—for every single document structure contained in a catalog—is tedious, error prone and costly.

The goal of the *Catalog Description Language (CDL)* is to provide a declarative specification language for product catalogs, from which these structure-dependent artifacts can be generated. This increases the level of abstraction of catalog management system development, by using generation to replace stereotype implementation activities.

Target System The Catalog management systems generated from CDL specifications comprise two types of code: generic framework code, which implements functionality common to all catalog management systems, and catalog specific code, which gets generated from CDL specifications.

As suggested by the Generation Gap pattern [28], inheritance is used to separate generic framework code (which resides in base classes) from catalog specific, generated code (which resides in generated subclasses).

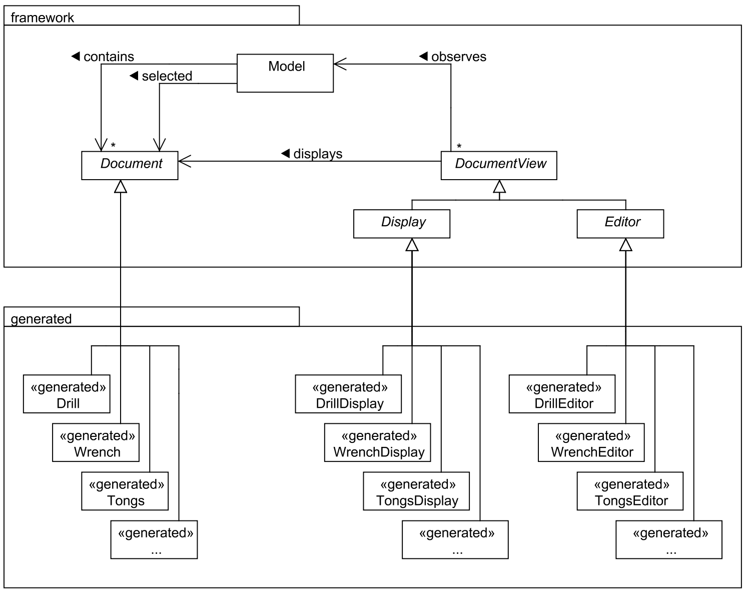


Fig. 5. Framework architecture

The catalog management systems use a simple Model View Controller [29] architecture (compare Figure 5): A central *Model* stores all documents of a catalog. *DocumentViewers* (that serve both as viewers and controllers) are used to display and edit documents. Common functionality resides in the abstract base classes *Document*, *Display* and *Editor* in the *framework* package.

The document structure specific code resides in classes in the *generated* package, which derive from the abstract base classes. For each document family specified in a CDL document, a document class, a display class and an editor class are generated.[7]

Figure 6 shows the different conceptual layers of the catalog management system ordered by their level of abstraction. The higher an artifact appears in the figure, the higher its specialization and potential fitness to solve a domain problem and thus the lower its reusability to other problems in the domain.

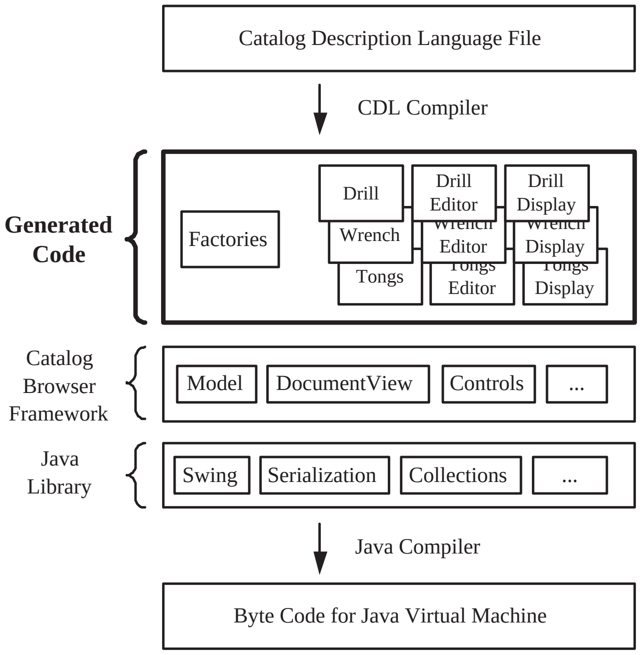


Fig. 6. CDL stack

[7] The current version uses serialization as a generic persistence mechanism and does not support data export. In a future version, persistence and export code will also be generated from document structure specifications.

Initial Language Version Listing 1.1 shows an exemplary specification[8] for a tool catalog written in the initial version of CDL. The file specifies document structures for two product families: *Wrenches* consist of a single multi-line text field *Description*, whereas *Drills* comprise one single-line text field *Headline* and two multi-line text fields *Description* and *Shipment*. The captions depict field labels displayed in editor forms.

Due to space constraints, the complete specification of CDL and the evolution operations applied during its evolution have been omitted. Refer to [30] and [31] for a complete reference of the implementation and evolution of CDL.

```
version 1
Wrench {
  multiline  Descript   caption "Description";
}

Drill {
  singleline Headline   caption "Family";
  multiline  Descript   caption "Description";
  multiline  Shipment   caption "Shipment Info";
}
```

Listing 1.1. CDL file in version 1

6.2 Evolving the Language

As is typical for incremental development, the first language version only comprises a small set of core language elements. Instead of designing the complete language up-front, we will grow it in small steps. This saves us from the effort and cost of performing a domain and variability analysis for our DSL. Furthermore, as we employ the first version of CDL to create catalog specifications, our understanding of the domain grows and we get feedback on our language design. Based on this feedback, we can make founded decisions on how to evolve the language.

In the following, we present two exemplary evolution steps: A relatively simple transformation that changes the concrete syntax, and a more complex transformation that restructures the language in a non-local way.

Local transformation As a first change, we decide to make the concrete syntax of CDL more expressive, by adding the keywords *catalog*, *document* and *field* and encapsulating the documents of a catalog in curly braces. Since this change only affects the concrete syntax of our language and leaves its abstract syntax unchanged, no semantic rules have to be updated.

Listing 1.2 depicts the required evolution operations. Line 2 contains a Language Evolution Language statement that inserts the keyword *catalog* into the

[8] simplified due to space constraints

production *Cat* in the DSL Dictionary. *lbl* is the label of the new catalog keyword, *docs* is the label of the dictionary element before which the new keyword gets inserted. [9] The statements in lines 3-10 behave accordingly for the braces and remaining keywords.

These evolution statements offer a high level of abstraction to the DSL developer, since they transform both the DSL Dictionary and the syntax tree. Listing 1.3 shows the CDL file after transformation. The new keywords introduced by the evolution operations are depicted in bold font.

```
1  # Add catalog keyword and brackets
2  insert_lit_before(”lbl”, ”catalog”,”docs”,”Cat”);
3  insert_lit_behind(”open”, ”{”, ”label”, ”Cat”);
4  insert_lit_behind(”close”, ”}”, ”docs”, ”Cat”);

6  #Add document keyword
7  insert_lit_before(”label”, ”doc”, ”name”, ”Doc”);

9  #Add field keyword
10 insert_lit_behind(”label”, ”fld”,”type”,”Field”);
```

Listing 1.2. Evolution operations for version 2

```
version 2
catalog {
  document Wrench {
    multiline  field Descript  caption ”Description”;
  }

  document Drill {
    singleline field Headline   caption ”Family”;
    multiline  field Descript   caption ”Description”;
    multiline  field Shipment   caption ”Shipment Info”;
  }
}
```

Listing 1.3. CDL file in version 2: local change of concrete syntax

Non-local transformation At this stage of development, we receive the requirement that a catalog management system must support users that speak different languages. As a consequence, catalog descriptions must be extended to support field labels in multiple languages. We decide to extract the field captions from the field definitions in order to preserve readability in the presence of many languages.

[9] In Lever, every part of a DSL Dictionary is labeled—language evolution operations can thus refer to the DSL Dictionary elements they work on by their names.

Listing 1.4 shows the CDL file after transformation. Lines 11-19 have been created by the evolution operations. Now that the labels have been extracted into a captions region, further captions regions can be added for additional languages.

This evolution scenario is an example for a non-local restructuring. It cannot be specified completely using high-level Language Evolution Language Statements alone. Rather, statements from the low level grammar and word evolution languages have been used to perform this evolution step. [10]

```
 1  version 3
 2  catalog {
 3    document Wrench {
 4      multiline  field Descript;
 5    }

 7    document Drill {
 8      singleline field Headline;
 9      multiline  field Descript;
10      multiline  field Shipment;
11    }

13    captions english {
14      Wrench {
15        Descript ”Description”;
16      }

18      Drill {
19        Headline ”Family”;
20        Descript ”Description”;
21        Shipment ”Shipment Info”;
22      }
23    }
24  }
```

Listing 1.4. CDL file in version 3: non-local restructuring

7 Conclusion

DSLs are a promising approach to increase the productivity of software development through raising the level of abstraction and providing powerful generative techniques. However, DSLs are expensive to build and even more expensive to maintain. The concepts and implementation techniques presented in this paper allow a new style of DSL development and maintenance by incremental step-wise evolution. This strategy renders the critical task of domain analysis less time-consuming and critical and significantly reduces the costs of changing a DSL by

1. automatically transforming all existing words in previous versions of the DSL and

[10] The evolution script comprises about 20 evolution operations and has been left out of this paper for brevity.

2. providing prototypical tool-support for the adaptation of the DSL compiler.

As shown with the exemplary product catalog description language, DSL architects are enabled to introduce flexibility into the DSL as needed at any time. The key to this flexibility is the transformation tool Lever (language evolver) that itself implements a powerful DSL for grammar, word, and coupled transformation for the consistent manipulation of DSLs.

As shown in this paper, a tool like Lever contributes to the construction of more powerful DSLs that span several levels of abstractions because this can only be done realistically by structuring the compilation process into a sequence of generation steps with a corresponding set of DSLs. Building and maintaining such a sequence of DSLs without tool supported and coupled transformation of grammars and words seems highly impractical.

Clearly, this work leaves room for interesting future work. One open question is how to further automate the adaption of the compiler and other language processing tools according to chances of the language. Another question that we will further investigate in the future is the actual construction of realistically applicable multi-level DSLs with the tool Lever implemented as part of the work presented in this paper.

References

1. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. SIGPLAN Notices **35**(6) (2000) 26–36
2. Lientz, B.P., Bennet, P., Swanson, E.B., Burton, E.: Software Maitenance Management. Addison Wesley, Reading (1980)
3. STSC: Software Reengineering Assessment Handbook v3.0. Technical report, STSC, U.S. Department of Defense (March 1997)
4. Pigoski, T.M.: Practical Software Maintenance. Wiley Computer Publishing (1996)
5. Deursen, A., Klint, P.: Little languages: little maintenance? Technical report, Amsterdam, The Netherlands (1997)
6. Mernik, M., Heering, J., Sloane, A.: When and how to develop domain-specific languages. Technical Report SEN-E0517, CWI (December 2005)
7. Stützle, R.: Wiederverwendung ohne Mythos: Empirisch fundierte Leitlinien für die Entwicklung wiederverwendbarer Software. PhD thesis, Technische Universität München (April 2002) (Engl.: Reuse without Myth).
8. Lehman, M., Ramil, J.F., Wernick, P.D., Perry, D.E., Tursky, W.M.: Metrics and laws of software evolution - the nineties view (1997)
9. Chaitin, G.: Register allocation and spilling via graph coloring. SIGPLAN Not. **39**(4) (2004) 66–74
10. McIlroy, M.: Mass produced software components. In Naur, P., Randell, B., eds.: Software Engineering, Garmisch, Germany, NATO Science Committee (October 1968) 138–155
11. Panas, T., Löwe, W., Ašmann, U.: Towards the unified recovery architecture for reverse engineering. In: SERP'03. Volume 1., Las Vegas, NV, CSREA Press (June 2003) 854–860
12. Eisenecker, U.W., Czarnecki, K.: Generative Programmierung. Addison-Wesley, München (2002)

13. Visser, E.: A survey of strategies in program transformation systems. Electronic Notes in Theoretical Computer Science **57** (2001)
14. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Trans. Softw. Eng. Methodol. **14**(3) (2005) 331–380
15. Lämmel, R.: Grammar Adaptation. In: Proc. Formal Methods Europe (FME) 2001. Volume 2021 of LNCS., Springer-Verlag (2001) 550–570
16. Lämmel, R., Wachsmuth, G.: Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In: Proc. of the 1st Workshop on Language Descriptions, Tools and Applications (LDTA'01), publisher =. (April 2001)
17. Kort, J., Lämmel, R.: The grammar deployment kit - system demonstration. In: Proceedings of the Second Workshop on Language Descriptions, Tools and Applications (LDTA'02). Volume 65 of ENTCS., Elsevier Science (2002)
18. Staudt, B.J., Krueger, C.W., Garlan, D.: A structural approach to the maintenance of structure-oriented environments. In: SDE 2: Proc. of the 2nd software engineering symposium on Practical software development environments, New York, NY, USA, ACM Press (1987) 160–170
19. Garlan, D., Krueger, C.W., Lerner, B.S.: Transformgen: automating the maintenance of structure-oriented environments. ACM Trans. Program. Lang. Syst. **16**(3) (1994)
20. Banerjee, J., Kim, W., Kim, H.J., Korth, H.F.: Semantics and implementation of schema evolution in object-oriented databases. In: SIGMOD '87: international conference on Management of data, New York, NY, USA, ACM Press (1987) 311–322
21. Penney, D.J., Stein, J.: Class modification in the gemstone object-oriented dbms. In: OOPSLA '87, New York, NY, USA, ACM Press (1987) 111–117
22. Ferrandina, F., Meyer, T., Zicari, R., Ferran, G., Madec, J.: Schema and database evolution in the O_2 object database system. In: Proc. of the 21th Intern. Conf. on Very Large Data Bases, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1995) 170–181
23. Claypool, K.T., Jin, J., Rundensteiner, E.A.: Serf: schema evolution through an extensible, re-usable and flexible framework. In: CIKM '98, New York, NY, USA, ACM Press (1998) 314–321
24. : Jython Home Page. `http://www.jython.org/` (January 2003)
25. CWI: SGLR. `http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/SGLR` (January 2006)
26. Paakki, J.: Attribute grammar paradigms a high-level methodology in language implementation. ACM Comput. Surv. **27**(2) (1995) 196–255
27. W3C: XSL Transformations (XSLT). `http://www.w3.org/TR/xpath` (November 1999)
28. Vlissides, J.: Generation Gap. `http://www.research.ibm.com/designpatterns/pubs/gg.html` (December 1996)
29. Gamma, E., Helm, R., Johnson, R., Vissides, J.: Design patterns : elements of reusable object-oriented software. Addison Wesley (1995)
30. Juergens, E.: Evolutionary Development of Domain Specific Languages. Master's thesis, Technical University of Munich (March 2006)
31. Juergens, E., Pizka, M.: Tool Demonstration: The Language Evolver Lever. In Boyland, J., Sloane, A., eds.: Sixth Workshop on Language Descriptions, Tools, and Applications (LDTA). (April 2006)

Kumbang Modeler: A Prototype Tool for Modeling Variability

Hanna Koivu, Mikko Raatikainen, Marko Nieminen, and Tomi Männistö

Helsinki University of Technology
Software Business and Engineering Institute (SoberIT)
P.O. Box 9210, 02015 TKK, Finland
{Hanna.Koivu, Mikko.Raatikainen, Marko.Nieminen, Tomi.Mannisto}@tkk.fi

Abstract. Variability is the ability of a system to be efficiently extended, changed, customized, or configured for use in a particular context. Several methods of modeling variability have been reported. However, tool support is also needed to take full advantage of models. We describe a prototype tool called Kumbang Modeler. Kumbang Modeler enables modeling the variability of a software product family using Kumbang conceptualization and language. The study follows the design science methodology. A user-centered design was applied in the development of Kumbang Modeler, and light-weight usability tests in the evaluation. The usability tests show that a person with knowledge of Kumbang concepts is able to correctly model the variability of a software product family using Kumbang Modeler.

1 Introduction

Variability is the ability of a system to be efficiently extended, changed, customized, or configured for use in a particular context [1]. As a successful means of managing variability, a *software product family* approach has emerged [2]. A software product family refers to a set of *product individuals* that reuse the same software assets and have a common structure called a software product family architecture [3]. The assets and the architecture of a software product family contain variability. This variability is resolved and, typically, product-specific software is also developed in order to derive the different product individuals of a software product family [4]. In a configurable software product family, the product individuals are constructed entirely on a basis of existing software only by resolving the variability [5,6].

A configurable software product family can contain a great amount of variability. In addition, this variability includes, for example, traceability relations and constraints [7]. Therefore, capturing and managing variability is challenging. Different variability modeling methods have emerged for variability management. When variability is expressed rigorously, such as in models with adequate rigor, product derivation can benefit from tool support. Tools can deduce consequences, check conformance to the model, and show when all variability is bound, for example [8]. Modeling also benefits from tools. Tools can hide the details of the modeling language and tools can provide a viewpoint to model such that the model is only partially visible and easily navigatable. Hence, product expects can express variability without considering the syntax details of a particular language A modeling tool should also make it easier to demonstrate variability modeling to companies that could benefit from it, as well as making it more approachable.

In this paper, we describe Kumbang Modeler, which is a prototype modeling tool for Kumbang. Kumbang [9] is a domain ontology for modeling the variability in software product families. Kumbang includes concepts for modeling variability, from the point of view of the component and feature structures. The language that utilizes Kumbang conceptualization is likewise called Kumbang. The modeling tool is an Eclipse plug-in [10] for creating and editing Kumbang models. The tool stores models using the Kumbang language. In the study, we followed the design science methodology [11]. For developing the tool, we applied a user-centered design, and, in particular, the personas method [12,13]. The tool was tested for feasibility and in two lightweight usability tests.

The rest of the paper is organized as follows. In Section 2, we describe the research methods. In Section 3, we give an overview of Kumbang. Section 4 introduces Kumbang Modeler. In Section 5, we describe the validation of Kumbang Modeler. Section 6 discusses our experiences in developing Kumbang Modeler. Section 7 outlines related work. Section 8 draws conclusions and provides some directions for future work.

2 Method

The research followed the design science methodology [11]. The construction is a prototype Kumbang Modeler. The objective of the prototype was to provide the user with the ability to be able to create and edit Kumbang models through a graphical user interface. The models made using Kumbang Modeler should be saved to text files using the Kumbang language. The study built on existing Kumbang concepts, without changing them.

The development of Kumbang Modeler followed a user-centered design. The methods used were *Goal-Directed Design*, and especially *Personas* [12,13]. However, users and context, which are central in user-centered design [14], could not be studied in practice, e.g., in a case study. This was because Kumbang Modeler is a new kind of product and not used anywhere. Therefore, different use scenarios and characteristics of potential users were explored, mainly on the basis of results reported in the literature. The objective was to achieve a better understanding of the effective use of a modeling tool in an industrial setting, and the skill requirements for users. In addition, the feasibility of a user-centered design without actual users was assessed.

Kumbang Modeler was tested for feasibility and in lightweight usability tests. In the feasibility test, different models were developed and their correctness was validated. The lightweight formative usability tests were carried out with two different users. Both users had experience with software product families. The users had not used or even seen Kumbang Modeler before the test. One user did not have knowledge of Kumbang, while the other was familiar with Kumbang concepts and language. The tests took roughly an hour and were done in an office room with a PC. Both tests were recorded with a video camera and screen capture software. The users were interviewed before and after the tests.

3 Kumbang Overview

Kumbang [9] is a domain ontology for modeling variability in software product families. Kumbang differentiates between a *configuration model*, which describes a family and contains variability, and a *product description*, which is a model of a product individual derived from a configuration model by resolving variability. The elements in a configuration model are referred to as *types*, while the elements in a product description are referred to as *instances*.

Kumbang includes concepts for modeling variability from both a structural and feature point of view. More specifically, the modeling concepts include *components* and *features* with *inheritance structure* and *compositional structure*, *attributes*, the *interfaces* of components and *connections* between these, and *constraints*. A compositional structure is achieved through the concepts of a *subfeature definition* and *part definition* that state what kinds of parts can exists for a feature or component, respectively. Constraints can be specified within components and features. Implementation constraints are a special class of constraints between features and components.

The semantics of Kumbang is rigorously described using natural language and a UML profile. A language based on the Kumbang ontology, likewise called Kumbang, has been defined. Kumbang has been provided with formal semantics by defining and implementing a translation from Kumbang to WCRL (Weight Constraint Rule Language) [15], a general-purpose knowledge representation language with formal semantics.

A tool called Kumbang Configurator, which supports product derivation for software product families modeled using Kumbang, has been implemented [8]. Kumbang Configurator supports a user in the configuration task as follows: Kumbang Configurator reads a Kumbang model and represents the variability in the model in a graphical user interface. The user resolves the variability by entering her requirements for the product individual: for example, the user may decide whether to include an optional element in the configuration or not, to select attribute values or the type of a given part, or create a connection between interfaces. After each requirement entered by the user, the Kumbang Configurator checks the consistency of the configuration, i.e., whether the requirements entered so far are mutually compatible, and deduces the consequences of the requirements entered so far, e.g., automatically choosing an alternative that has been constrained down to one; the consequences are reflected in the user interface. The consistency checks and deductions are performed using an inference engine called smodels [15] based on the WCRL program translated from the model. Once all the variation points have been resolved and a valid configuration thus found, the tool is able to export the configuration, which can act as an input for tools used to implement the software, or used for other purposes.

4 Kumbang Modeler

This section introduces Kumbang Modeler, a prototype tool for creating Kumbang models. Kumbang Modeler has been implemented as a plug-in for the Eclipse Platform [10]. First, a short introduction to Eclipse will be given below, then Kumbang Modeler is described in terms of architecture, user interface, and usage.

4.1 Eclipse

Eclipse [10] is an integrated development environment popular among Java developers [16]. Eclipse began as an open source IDE tool for Java development, but has been extended to a multi-purpose development environment via plug-in extensions. Eclipse plug-ins are currently very popular [17].

Eclipse's development environment is called a *workbench* [18]. A user sees a workbench as one or several windows. Each window contains a menu bar, a toolbar, and one or more *perspectives*. A perspective defines what is included in the menus and toolbar. The perspective also defines a default layout, which can be changed or reloaded to undo changes. A perspective is also a container that defines the initial group and layout of a group of *editors* and *views*. An editor or a view contains the actual user interface elements. Plug-ins can consist of any of the element such as views, editors, menus, or perspectives.

4.2 Kumbang Modeler Architecture

The main elements of the Kumbang Modeler architecture are the model layer, the controllers layer, and the user interface layer. A Kumbang model is represented as Java objects at the model layer and can be imported from or exported to a text file. The controllers layer combines some display information with he model objects, provides ways to change the model and updates these changes at the graphical user interface layer. The user interface elements are at the graphical user interface layer. Kumbang Modeler reuses the model and parser from Kumbang Configurator [8].

4.3 Kumbang Modeler User Interface

User interface elements specific to Kumbang Modeler are a perspective, an editor, and three views. In addition, Kumbang Modeler uses two standard Eclipse views. The user interface design was guided by the Eclipse User Interface Guidelines [19].

The perspective for Kumbang Modeler comprises six different areas, depicted by letters a-f in Figure 1. The perspective is automatically opened when a file containing a Kumbang model is opened or a new Kumbang model is created.

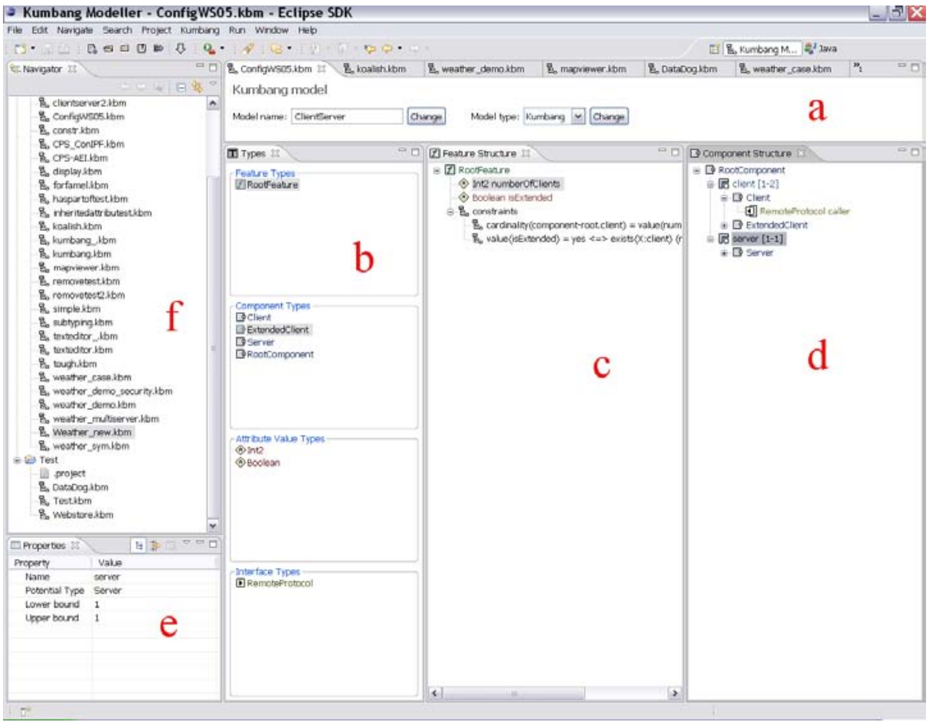
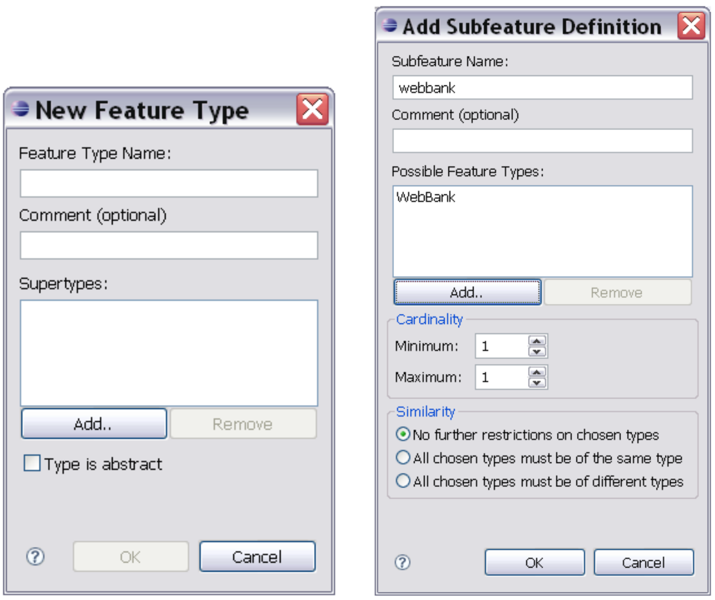


Fig. 1: The perspective for Kumbang Modeler: a) the editor area b) the type view c) the feature view d) the component view e) the properties view f) the navigator view

The editor area (Figure 1, a) shows the currently active editor and enables switching between open editors by selecting the model from the tabs. The editor area displays the model name and type; all other information is shown in the views. This gives the user more control over the user interface, as views can be resized, moved, and closed freely. The editor takes care of opening and saving the model. Several editors can be open at the same time.

The three views peculiar to Kumbang are *a type view*, *a feature view* and *a component view* (Figure 1 b, c and d, respectively). The type view lists currently available types. Kumbang has feature, component, interface, and attribute value types. The feature and component views show the compositional hierarchies. The hierarchies form trees with one root. The tree is composed using the sub-feature definitions within the features types and the part definitions within the components types. In addition, constraints are added to the feature and component types in the feature and component views. Implementation constraints between the feature hierarchy and the component hierarchy are placed in the feature view.

The *properties* and *navigator views* are standard views in the Eclipse IDE. The properties view (Figure 1, e) shows additional information on the currently


New Feature Type

Feature Type Name:

Comment (optional)

Supertypes:

☐ Type is abstract

Add Subfeature Definition

Subfeature Name: webbank

Comment (optional)

Possible Feature Types: WebBank

Cardinality

Minimum: 1

Maximum: 1

Similarity

[✓ No further restrictions on chosen types]
All chosen types must be of the same type
All chosen types must be of different types

(a) (b)

Fig. 2: Two Kumbang Modeler dialogs: a) a dialog for adding a new feature type; b) a dialog for a subfeature definition.

selected elements. The navigator view (Figure 1, f) shows Eclipse projects and files.

Finally, Kumbang Modeler contains several dialogs that are needed when editing a model. The archetypes of the dialogs are shown below in the section describing the usage of Kumbang Modeler.

4.4 Usage

The available types, such as different feature and component types, of a model are created using dialogs. Figure 2(a) shows a dialog for creating a new feature type; similar dialogs are used for other types, although the exact fields are peculiar for the respective type. The dialogs are needed in order to set all details of the specific type. For example, a new feature type needs a name, possible supertypes, specification if the type is abstract, and an optional comment. The same dialogs that are used for creating new types are used to display and change existing types. Right-clicking in any of the views peculiar to Kumbang opens a menu from which an option for the dialog for creating new types can be selected. Alternatively, the dialog can be opened and new types created while defining compositional structures other than root, as described below.

The compositional structure of the features and components needs to proceed from the root to the leafs. When a type is dragged from the type view to an empty structural view of that type, the type becomes the root; or, when there is no root set, an existing type can be selected to be the root from a list of all possible types. Consequently, only existing type can be selected to be a root.

When the root is set, the other types can be added to the compositional structure through their respective definitions. A type can be added to the structure by dragging. Alternatively, an existing type can be selected or a new type can be created for the compositional structure during the construction of the definition. Attributes and interfaces are attached to the structure using the concept of definition, similar to the way the compositional structure is constructed. A dialog is always needed to determine the necessary information when adding a type to a structure. Figure 2(b) shows an example dialog for a subfeature definition. If a type is dragged to the tree, those values that have feasible default values are pre-filled. For instance, the cardinality of definitions has a default value of one-to-one, and the name is derived from the type, but the direction of an interface definition has no sensible default value.

The constraint language of Kumbang [20] combines predicates with Boolean algebra. A constraint can be very complex; hence, there is no simple way to manage them. In addition, flexibility is required in constructing the constraints, since length cannot be determined beforehand. The approach taken in Kumbang Modeler splits constraints into parts that can be constructed separately. There are two kinds of basic parts: *expressions*, which are predicates or functions, and *operators*, which combine the expressions or implicate relationships between the expressions. These parts are shown in a list, which is expanded every time a part is added, as seen in Figure 3(a). When a new expression is added, an expression type must first be chosen (Figure 3(b)). A similar dialog is also used for choosing operators. Each expression has a special dialog for defining the details.

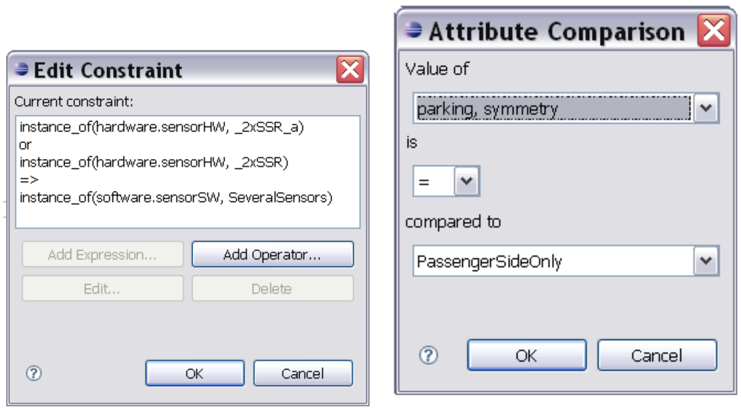
5 Validation of Kumbang Modeler

5.1 Feasibility to Produce Valid Models

We tested whether Kumbang Modeler is able to produce valid models based on Kumbang concepts and written in the Kumbang language. The models were syntactically correct and could be opened also in Kumbang Configurator. In addition, Kumbang Modeler can be used to open and modify various existing Kumbang test models.

5.2 Usability Evaluation of the Prototype

Kumbang Modeler was tested through lightweight formative usability tests with two different users by a predefined modeling task defined as a scenario. The first user, who knew Kumbang very well, had very little trouble making a model according to the scenario. She did have some suggestions for improving the user



Current constraint:

```
instance_of(hardware.sensorHW, _2xSSR_a)
or
instance_of(hardware.sensorHW, _2xSSR)
=>
instance_of(software.sensorSW, SeveralSensors)
```

Value of: parking, symmetry
is: =
compared to: PassengerSideOnly

Fig. 3: A dialog for editing a constraint and a dialog for choosing an expression to a constraint.

interface, however. Most of these proposals were implemented before the second usability test. The second user had no previous knowledge of Kumbang before participating in the test. He had trouble understanding the need for relation between types and definitions used for compositional structure. This made him very frustrated; when he wanted to add a feature to the model, he did not understand why he had to make both a feature type and a subfeature definition to achieve this. However, he was able to produce an acceptable model. Table 1 summarizes the main usability changes made to Kumbang Modeler on the basis of the usability tests.

Problem	**Action taken**
The icon used in an interface definition did not show whether the interface was provided or required.	Instead of a single interface icon, different icons was designed for a required and provided interface.
Subfeature definitions could not be moved in the feature structure.	Subfeature definitions were made draggable.
User was irritated with having to name both types and definitions.	Lower case version of the type name was made to be the default definition name.
User was irritated with having to do too many steps when wanting to make a definition with a type that did not exist yet.	Several steps were combined into one.

Table 1: Changes made after the feedback from the usability tests

6 Discussion

6.1 Feasibility of Tool Support for Variability Modeling

Kumbang Modeler seems feasible for modeling variability in software product families. In addition, Kumbang Modeler seems to have advantages over writing a model by a text editor. For example, the produced models are syntactically correct, and the tree structure seems to make navigating within the hierarchy easier.

Kumbang Modeler was developed as a plug-in for Eclipse, which seems to be a practically applicable platform for such a modeling tool as Eclipse is currently a popular development environment. In addition, different plug-in extensions are also relatively widely used and easy to install. Many developers are thus familiar with Eclipse as a development environment, and with plug-in extensions for Eclipse.

6.2 Validation of Kumbang Modeler

Two tests for Kumbang Modeler were carried out: the test of the validity of the models produced and the usability tests. However, these tests have weaknesses. First, Kumbang Modeler does not check anything other than the syntactical correctness of a produced model. However, we are currently implementing advanced checks for Kumbang Modeler. The advanced checks ensure, for example, that a model does not contain cycles in the inheritance, part, or subfeature structures; a model contains all the references needed, such as the type declarations for the types used in a part definition, and at least one configuration can be found such that all interfaces can be connected; for every required interface, a provided interface exists; and constraints are not in conflict with each other. Second, the usability tests were lightweight and were not carried out using a real product. Hence, more usability tests are required in industrial settings.

6.3 Feasibility to use Kumbang Modeler to Model Variability

Kumbang Modeler seems to be feasible for product experts to express the variability of a software product family as Kumbang models. The user who knew Kumbang concepts was able to use the tool without difficulties. The user seems to be, however, required to have some knowledge of Kumbang and software product families, as the second usability test showed. Nevertheless, a thorough understanding of Kumbang syntax and semantics did not seem to be needed. The requirement of understanding Kumbang concepts is not necessarily a problem, since the tool is meant for highly specialized use. However, more tests are needed, as argued above.

The difficulties the other user had with producing a model using Kumbang Modeler seemed to be more related to Kumbang concepts than Kumbang Modeler as a tool. The user had no previous knowledge of Kumbang and was, in fact, used to modeling software product families differently. Especially troublesome were those concepts that are not widely used in other modeling approaches. Three issues especially caused difficulties: the type and instance differentiation, the part and subfeature definitions in the compositional structure, and terminology. These are discussed in more depth below.

First, many feature modeling methods, for example, do not differentiate between types and instances. However, in Kumbang they are used in order to distinguish between a family model that contains variability and an instance model in which variability is resolved, enable several instances of the same type, and enable feature type reuse in different places of the model [21]. In addition, software product family engineering distinguishes between family and instance, e.g., in a form of family and instance development processes, or reusable and reused assets. Consequently, differentiating between types and instances seems reasonable.

Second, despite the fact that subfeatures and the compositional structure of components are used in most modeling methods, the subfeature and part definitions, which are slightly different in Kumbang, caused difficulties. The subfeature and part definitions are regarded as required in Kumbang [21]. However, especially in simple structures, such as in a subfeature structure without variability, it seems that in many cases, default values can be used for the details of the subfeature definitions. The user interface was simplified, e.g., by making a lowercase version of the type name the default name of the part definition. Although the subfeature definitions, for example, can be simplified in the user interface by using default values, they are still needed in order to express complex variability.

Third, the terminology of Kumbang Modeler was confusing. However, in software product family engineering the same term is often used to refer to different concepts or several terms are used to refer the same concept. For example, feature modeling methods do not terminologically distinguish between feature types and feature instances or product derivation can be also called instantiation, deployment or configuring. Hence, the terminology in general is ambiguous and not established. A person used to one terminology can get confused when she needs to use another terminology.

6.4 Variability Modeling

Issues concerning the nature of variability arose during the study. For example, Kumbang uses constructs that can be used to model complex variability. However, much of the variability in usability tests was so simple that using Kumbang constructs meant inserting information that was laborious, and default values would have been feasible. In order to enhance tool support, the nature of variability in software in terms of, for example, the amount and complexity of variability needs to be studied in more depth. This could then be used to develop tools that meet the actual requirements for modeling variability. For example, syntactic sugar on top of modeling concepts could be developed in order to hide complex structures. However, the rigor of the models should not be lost. The models should be based on a well-founded conceptual foundation.

6.5 User-Centered Design

We faced problems with the user-centered design approach during the development of Kumbang Modeler. The users of Kumbang Modeler do not exist and, hence, cannot be studied. We tried to study the literature in order to capture, e.g., the skills of potential users, but little is reported in the literature. Another option would have been to carry out a user study of software product family engineers in general, but this was considered to be beyond the scope of this study.

Goal-Directed Design considers necessary-use scenarios to be less important than daily-use scenarios. However, Kumbang Modeler is mainly a prototype tool and thus the threshold for using it for modeling should be low. Therefore, the creation of new models is just as important in Kumbang Modeler as modifying existing ones, although only modifying a model can be considered a daily use scenario. Therefore, Goal-Directed Design was not directly applicable in Kumbang Modeler design.

The usability tests brought about the same problem as with the overall development of the tool, namely a lack of real users who would use the tool in an actual, industrial environment. However, we assumed that such users could have three kinds of knowledge: knowledge of the specific configurable product family, configurable product family concepts, and modeling concepts. Since configurable product families are hard to find, in usability tests we used two kinds of user: both had knowledge of configurable product family concepts and one knew Kumbang.

7 Related Work

Several software variability modeling methods have been developed in addition to Kumbang, such as feature modeling [22], orthogonal variability modeling [23], and COVAMOF [24]. Different kinds of tools have been developed for the modeling methods; a review of a set of tools is provided in the ConIPF methodology [25].

In addition, there are variability modeling tools that are not used for software products. Instead, the tools are originally meant for modeling mechanical and electronic products. Examples of such tools are the Wecotin [26] and EngCon [27] modelers.

Tools can be also used in other phases of the software life cycle. In ConIPF methodology [25], tools are used for requirements engineering, modeling, configuring, realization, and software configuration management. Different tools can be used in different phases. Kumbang currently has tool support for the modeling and configuration phases. Different tools or new tools for Kumbang need to be developed for the other phases of development.

8 Conclusions

In this paper we described Kumbang Modeler, which is a tool for modeling the variability of a software product family. Modeling is based on Kumbang conceptualization. Hence, Kumbang Modeler enables modeling both from a structural and feature point of view. The study followed design science methodology. We applied a user-centered design in developing Kumbang Modeler; more specifically, the Goal-Directed Design and Personas methods. Kumbang Modeler was tested for feasibility to produce correct models and in lightweight formative usability tests.

The results showed the feasibility of modeling variability with Kumbang Modeler. At least some knowledge on the applied Kumbang variability concepts is required to use the tool. We faced problems with the user-centered design because actual users were not available. The usability tests, nevertheless, showed that despite the fact that variability can be modeled with the existing methods, more studies are needed to show that modeling is efficient and convenient. For example, much of the variability can be simple and details of more complex constructs to model variability can then be hidden or default values can be used. However, modeling also seems to need complex structures. In addition, in order for, e.g., tool-supported derivation to be possible, modeling should be based on rigorous foundations.

Kumbang Modeler provides the missing tools support for Kumbang. That is, with Kumbang Modeler, the captured variability of a software product family can be modeled, whereas with the existing Kumbang Configurator, expressed variability can be bound during product derivation. Hence, other tools, such as a generator, are still needed.

Acknowledgements

The authors acknowledge the financial support of Tekes, the Finnish Funding Agency for Technology and Innovation.

References

1. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Software — Practice and Experience **35** (2000)
2. Weiss, D.M., Lai, C.T.R.: Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
3. Clements, P., Northrop, L.M.: Software Product Lines: Practices and Patterns. Addison-Wesley (2001)
4. Bosch, J.: Design and Use of Software Architecture. Addison-Wesley (2000)
5. Bosch, J.: Maturity and evolution in software product line: Approaches, artefacts and organization. Lecture Notes in Computer Science (Proc. of SPLC2) **2379** (2002) 257–271

6. Männistö, T., Soininen, T., Sulonen, R.: Configurable software product families. In: ECAI 2000 Configuration Workshop, Berlin. (2000)
7. Thiel, S., Hein, A.: Modeling and using product line variability in automotive systems. IEEE Software **19**(4) (2002)
8. Myllärniemi, V., Asikainen, T., Männistö, T., Soininen, T.: Kumbang configurator—a configuration tool for software product families. In: IJCAI-05 Workshop on Configuration. (2005)
9. Asikainen, T., Männistö, T., Soininen, T.: Kumbang: A domain ontology for modelling variability in software product families. Advanced Engineering Informatics **21**(1) (2007)
10. Eclipse Foundation: Eclipse platform. http://www.eclipse.org/ (2006) Visited December 2006.
11. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. MIS Quarterly **28**(1) (2004)
12. Cooper, A.: The Inmates Are Running the Asylum. Macmillan Publishing Co. Inc. (1999)
13. Cooper, A., Reimann, R.: About Face 2.0: The Essentials of Interaction Design. John Wiley & Sons, Inc. (2003)
14. ISO/IEC: 9241-11 ergonomic requirements for office work with visual display terminals (vdt)s - part 11: Guidance on usability (1998)
15. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002)
16. Goth, G.: Beware the march of this ide: Eclipse is overshadowing other tool technologies. IEEE Software **22**(4) (2005) 108–111
17. Murphy, G.C., Kersten, M., Findlater, L.: How are java software developers using the eclipse ide? IEEE Software **23**(4) (2006) 76–83
18. Eclipse 3.2 Documentation. http://help.eclipse.org/help32/index.jsp (2006) Visited December 2006.
19. Edgar, N., Haaland, K., Li, J., Peter, K.: Eclipse user interface guidelines, v. 2.1 (2004) http://www.eclipse.org/articles/Article-UI-Guidelines/Index.html. Visited December 2006.
20. Asikainen, T.: Kumbang language, technical report (2007, to appear)
21. Asikainen, T., Männistö, T., Soininen, T.: A unified conceptual foundation for feature modelling. In: Proceedings of the 10th International Software Product Line Conference. (2006)
22. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, ADA 235785, Software Engineering Institute (1990)
23. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer (2005)
24. Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J.: Covamof: A framework for modeling variability in software product families. In: Proceedings of Software Product Line Conference (SPLC). (2004) 197–213
25. Hotz, L., Wolter, K., Krebs, T., Deelstra, S., Sinnema, M., Nijhuis, J., MacGregor, J.: Configuration in Industrial Product Families. IOS Press (2006)
26. Tiihonen, J., Soininen, T., Niemelä, I., Sulonen, R.: A practical tool for mass-customising configurable products. In: In Proceedings of International Conference on Engineering Design (ICED 03). (2003)
27. Hollmann, O., Wagner, T., Guenter, A.: Engcon - a flexible domain-independent configuration engine. In: Configuration Workshop at ECAI-2000. (2000)

Variability Management and Compositional SPL Development

Jilles van Gurp

Nokia Research Center, Helsinki, Finland
jilles.vangurp AT nokia.com

This position paper reflects on the implications for variability management related practices in SPL development when adopting a compositional style of development. We observe that large scale software development is increasingly conducted in a decentralized fashion and on a global scale with little or no central coordination. However, much of the current SPL and variability practices seem to have strong focus on centrally maintained artifacts such as feature and architecture models. We conclude that in principle it should be possible to decentralize these practices and identify a number of related research challenges that we intend to follow up on in future research.

Introduction

Over the past ten years software product line (SPL) conferences and related workshops have established SPL research as a new discipline in the broader field of software engineering. We, and others, have been contributors to this field with publications on software variability management [1, 2] and involvement in earlier workshops [3]. In more recent work, we have published about compositional development of software products [4]. Compositional development decentralizes much of the traditionally centralized activities in an integration oriented SPL, including requirements management, architecture and design. These activities are pushed down to the component level.

So far, Moore's law has accurately predicted the exponential growth of transistor density on chips and software developers seem to have matched this growth with a similar growth in software size (generally measured in lines of code). Decentralization of development activities allows development to scale better to such levels. This scalability is required to create more products that integrate a wider diversity of components and functionality. Making software products that support an ever wide range of functionality is necessary in order to differentiate in the market.

Unfortunately, decentralization has far reaching consequences for SPL methodology and tooling. A common characteristic of many of the currently popular SPL methodologies is the use of centrally maintained feature models that describe the variability in the SPL. For example, much of the SPL specific tooling depends on such models. This includes build configuration tools; requirements management tooling and product derivation support tools. Additionally, processes and organizations are generally organized around these tools and models.

In a compositional approach where development is decentralized and different components are developed by different people, business units and organizations using different methodologies and tools, these approaches break down. Products are not created by deriving from a central architecture and feature model but by combining different components and writing glue code.

The intention of this position article is to reflect on this topic and identify new research issues. It seems that much of the current research is conducted under a closed world assumption where one central organizational entity is in charge of overall design, management and governance of the SPL software. We believe this assumption to be flawed. The reality on the ground is quite different. Increasingly software companies are collaborating either directly or through open source projects on software assets that they have a shared interest in. It seems that it is almost impossible to develop software these days without at least some dependencies on external software. Additionally, in large companies software development is distributed throughout the organization. Consuming software from a business unit on a different continent poses very similar challenges to partnering with a different company.

In the remainder of this article we first briefly introduce the topic of compositional development before reflecting on what that means for variability management and finally reflecting on some future research topics related to that.

Compositional development

Compositional development might be interpreted as a move back to the COTS approaches popular in the past decade. In those days, it was suggested that companies would either buy components developed by other components or use in house developed components from a reusable component base. These approaches fragmented in roughly four directions over the past decade:

- **Integrated Platforms**. In this approach, one vendor offers a fully integrated software platform complete with tools, documentation, vast amounts of reusable software and consulting. Examples of companies that provide such vertical stacks of software are IBM, Oracle and Microsoft. While successful, this approach is mostly limited to the domain of enterprise systems. Characteristic of this domain is that most systems built are one of a kind.
- **SPLs**. For other domains than enterprise systems (e.g. embedded software), SPLs have emerged as a successful way to develop a platform in house and use that to build software products. Unlike enterprise software, most embedded software products are not one of a kind. Product lines for embedded software also tend to be highly specialized for the domain (e.g. mobile phones; audio visual equipment; medical equipment). Within those domains, the product line aims to support a wide range of products.
- **'True' COTS**. The vision in the nineties was not SPLs or huge vertical stacks of technology but a market of component vendors whose products

could be combined by product developers. Except for a few limited domains (e.g. GUI components), this market never emerged [5]. However, this market is comparatively small. Problems with respect to ownership of source code; interoperability; documentation and support are usually cited as the causes for this.

- **Open source**. From the mid nineties a vast amount of software has been released in the form of open source. Currently there are tens of thousands of active projects releasing high quality software. Most commercial software development, including embedded software development, now depends on substantial amounts of this software. Very few commercial software companies are 100% open source though. It seems many companies have a small layer of differentiating software & services that are not open source.

Of course, these approaches overlap. For example, several SPL case studies have been published for enterprise systems. Additionally using COTS in combination with either SPLs or enterprise platforms is quite common. In fact, most of the COTS companies seem to make components that are specialized for a particular platform. Finally, open source is important for COTS, integrated platforms and SPLs. Compositional development of products involves combining elements from all of these approaches and is certainly not about just COTS.

In compositional development, development teams of components or subsystems operate with a higher degree of autonomy then they would in a SPL organization. Identification of key requirements and design solutions is largely the responsibility of these teams. They interact with developers of other components they depend on and with developers of components (or products) that depend on them. However, the central coordination of this communication is absent.

The rationale here is to bring the decision processes as close to where it has an impact; and also to where the domain and technical experts operate. This is a quite different working model from the traditional one where a group of seniors decides together with the major stakeholders on design, requirements and other issues.

The problem with integrated SPL model is that it does not scale to the current industrial practice where software systems spanning multiple millions of lines of code are now the rule, not the exception. Managing the design and architecture of such software centrally is extremely difficult. The amount of people with a detailed enough knowledge of the software is very low in such companies. Additionally these people tend to be very busy and are generally very hard to replace.

In practice, this means that as software size grows, decision makers at the top are increasingly detached from the design details of the software. In other words, it disqualifies these individuals for making the technical decisions they should be making. The logical, and in our view inevitable, approach is to stop trying to take most of these decisions centrally.

A useful analogy here might be that of the communist era planned economies vs. the capitalist free market system. Making government level decisions about when, where, and how to move a few tons of tomatoes is obviously nonsensical to proponents of the latter. Yet this is exactly what happened in the strictly hierarchical organized planned economies leading to obvious issues such as one half of the country having a shortage of tomatoes and another half having tons of tomatoes rot in some central deposit. Similarly, detailed decisions about design and features are best left to the experts working on the actual software and any depending stakeholders.

Variability Management

Software variability is the ability of software assets to be extended, customized, configured or otherwise adapted. In SPLs, the intention is to have a set of reusable assets and architecture as well as a means to create software products from those. In other words, the reusable assets and architecture feature a degree of variability that is put to use during product creation.

In line with research from others at the time, our earlier work on variability management identified feature models as a means to identify so-called variant features in the requirements; and also as a means to plan the use of variability realization techniques to translate the variant features into variation points (i.e. concrete points in a software system where there is an opportunity to bind variants to the variation points during product creation).

More recent research, has focused on (partially) automating and supporting this process; formalizing the underlying models (e.g. using the UML meta model); tool support; etc. Some of these approaches are now used successfully in industry. A small commercial tool and support community is emerging. E.g., Big Lever (www.biglever.com) can support companies with support and tooling when adopting a SPL approach. MetaCase (www.metacase.com) provides similar services. Additionally, various research tools integrate feature modeling support into popular IDEs. Clearly, these tools are useful and various case studies seem to confirm that.

However, all of them more or less depend on the presence of a centrally governed architecture and feature model. Introducing compositional development implies less central control on these two assets. Consequently, requirements analysis and architecture design activities are also affected.

Assuming that software development is fully decentralized, this means the following:

- New features or variant features are identified, prioritized and implemented locally rather than at a central level.
- Important architecture decisions with respect to component variability and flexibility are mostly taken without consulting a central board of architects.
- New variant features are not represented in centrally maintained feature models unless the updating of such models is either automated or enforced with (central) processes and bureaucracy. This may be hard or even impossible given differences between organizations & processes of the various software development teams involved.
- For the same reason, any tool mapping of such new variant features to software variation points is not updated. Such mappings are critical in e.g. build configuration tools.

Provided vs. required variability

Feature models may be regarded as descriptions of either required or provided features in a software system. Feature models of required features are the output of the software analysis process. They may be interpreted as specifications of the software or be used to guide the design process. On the other hand, feature models of provided features describe implemented software systems in terms of the features actually implemented in the software. Models of provided features may be of use for e.g. configuring software products derived from the platform. In theory, these models should be the same but in practice requirements constantly change and few software products actually conform to initial requirements specifications. In fact, most development on large software systems is software maintenance and concerns changes to both the software and its provided feature specifications.

A similar distinction can be made for architecture documents. While the words 'architecture document' suggest that software is developed according to the blueprints outlined in this document, a more popular use of architecture documentation tools seems to be to document the design of already implemented software. This type of documentation is generally used to, for example, communicate the design to various stakeholders. Additionally, models described in an architecture description language may be used to do automated architecture validations; simulations; or system configuration.

When using a build configuration tool based on feature models, developers select existing implementations of features or variant features. In other words, they make use of a model of the provided variability. The tool in turn needs to map the feature configuration to variation points in the implementation artifacts. In other words, it has an internal model of the provided variation points in the software architecture.

This distinction of provided vs. required variability is highly relevant because we observe that much of the SPL tooling is more related to provided rather than required variability. De-centrally developed components may not conform to a centrally pre-defined model of required features but they certainly do provide features that may be described. Similarly, these components do not realize a pre-defined central architecture but may still provide explicit variation points. There is nothing inherently central about either feature models or architecture.

Research Issues & Concluding remarks

This article observes that there is a trend to decentralize software development in the current software industry and that this raises issues with respect to SPL development, particularly where it concerns the use of centrally defined feature models, architecture models, and related tooling. Fundamentally, this centralized/top down style of software development is not compatible with the bottom up style development seen across the industry and we foresee that this centralized approach will not continue to scale to the required levels. Already, the incorporation of de-centralized elements is evident in the increasing popularity & use of open source components, and also in publications such as Van Ommering.

From this observation, we explored a bit what it means to do decentralized compositional development and what that means for the centralized use of feature models and architecture models in current SPL development. An important conclusion we make is that most of the current tooling is focused around using feature models of provided features in a software system to configure provided variability points in a software architecture. We do not see any fundamental objections to continue doing that in a decentralized development model. Feature models of individual components may be provided and similarly the variability provided in these components may be described.

The above suggest that much of the tooling that currently exists for variability management may be adapted for use in a de-central fashion. Some potential research topics related to this that we intend to explore further in future work are:

- How to synthesize aggregated feature models and architecture models from the individual component level models given a component configuration.
- How to validate component configurations given incomplete feature & architecture information from components.
- How to deal with integrating components without formally documented features and variation (e.g. most open source software comes without such documentation).
- How to deal with crosscutting variant features that affect multiple, independently developed components. E.g., security related features generally have such crosscutting properties.

Some preliminary work related to this has already been done by amongst other Van Ommering [6] who wrote articles on KOALA and development issues related to compositional development. The trend, judging from KOALA, similar approaches and, also from the increased popularity of component frameworks such as standardized in OSGI, seems to be to address these issues with microkernel like architectures that explicitly requires components to state dependencies and interfaces.

References

[1] J. van Gurp, J. Bosch, M. Svahnberg, On the Notion of Variability in Software Product Lines, Proceeedings of WICSA 2001, 2001.

[2] P. Clements, L. Northrop, “Software Product Lines - Practices and Patterns”, Addison-Wesley, 2002.

[3] J. van Gurp, J. Bosch, Proceedings of First workshop on software variability management (SVM 2003), Groningen 2003.

[4] J. van Gurp, C. Prehofer, J. Bosch, Scaling Product Lines to new Levels: the
Open Compositional approach, submitted December 2006.

[5] B. Lang, Overcoming the Challenges of COTS, news @ SEI, 4(2) http://www.sei.cmu.edu/news-at-sei/features/2001/2q01/feature-5-2q01.htm, 2001.

[6] R. van Ommering, Building product populations with software components, proceedings of Proceedings of the 24rd International Conference on Software Engineering (ICSE 2002), pp. 255-265, 2002.

Variations in Model-Based Composition of Domains

Anca Daniela Ionita[1], Jacky Estublier[2], German Vega[2]

[1]Automatic Control and Computers Faculty, Univ. "Politehnica" of Bucharest,
Spl.Independentei 313, 060042, Bucharest, Romania
Anca.Ionita@mag.pub.ro
[2]LIG-IMAG, 220, rue de la Chimie BP53 38041 Grenoble Cedex 9, France
{Jacky.Estublier, German.Vega}@imag.fr

Abstract. As model driven engineering increases the level of abstraction, there are more possibilities to hide complexity and to introduce variability points. Variations and composition, which are usually complementary approaches, may be merged inside models and complement each other. The "domains", presented in this paper represent a coarse granularity reuse units. Domain variability is defined by models and the way domains are composed is also defined by a (composition) model. Therefore, variations are found at two levels, (1) inside domains (by choosing the appropriate model) and (2) in the composition mechanism itself (by defining the composition semantics).

Keywords: Model Driven Engineering (MDE), variability, composition, reuse

1 Introduction

For managing diversity, variation and composition are often considered two independent "schools of thought", but merging them has been identified as a necessity [1]. Variation, in product family approaches, supposes a top-down approach, based on an architecture where the elements, common to the whole family, and the "features" specific to each family member, are clearly identified. While planned variations are easy to perform (feature selection), unplanned evolution is difficult to address. Conversely, composition supposes a bottom-up approach, where existing units are selected and glued, in order to form an application. Composition is not limited to putting together components that perfectly fit together, but must also solve functional and non functional mismatches. This paper investigates the links between variation and composition, supported by Model Driven Engineering [2].

Composition may be seen as a variation mechanism because it allows building different combinations that constitute variations, and hierarchical composition [3] increases the variability potential. The current composition approaches use a rigid composition mechanism, which reduces both the reuse potential and the range of possible assemblies. Generally, variation only consists in changing method call parameters and/or in selecting the classes/components implementing the required interfaces. The composition logic and the variability are spread among components and cannot be changed without changing their code. To enhance flexibility, reusability and evolutivity, the control flow should not be the scattered in the components, but centralised, like in orchestration [4].

To improve both variation and composition, MDE raises the level of abstraction of the applications, allowing abstract composition mechanisms independent from technical details. In an MDE approach, the elements to compose are models, and composition means more than selecting and gluing pieces of code. A powerful model composition mechanism, which includes variability capabilities, is needed. Composition can be used to create large applications pertaining to product populations [5], but new concepts and mechanisms are required to support this complexity.

The paper discusses a new model-based composition mechanism, tailored for large units of reuse called "domains". Each domain is associated with a DSL (a metamodel) and an application in the domain is defined as a model written in this DSL (conform to the metamodel). Composing two applications (A and B) pertaining to two domains is reduced to composing their models (MA and MB). First, our approach composes the two DSLs (DSL_A and DLS_B) through the definition of new relationships between them and obtains a new DSL (DSL_AB). This (meta) composition may include some variability. Then, MA and MB are composed, while the composition model is expressed in DSL_AB, leaving MA and MB unchanged.

A development environment, called Mélusine has been realised and a correspondent methodology [7] has been defined for supporting this approach. Chapter 2 describes the variability introduced by the domain architecture, while chapter 3 presents the variation points related to domain composition through relationships. Both of them are exemplified on real examples used and reused in industrial applications.

2 Variations and Composition with Domains

2.1 Variability inside Domains

The *domains* presented in this paper have been designed to reuse heterogeneous components that do not know each other. The composition model is explicit, according to the domain specificity and to the application variability. The layered architecture allows the definition of variation points on multiple levels.

Domains are fully independent and autonomous: they do not have dependencies of any kind. A domain has a *domain specific language* and can interpret models conforming to it (see Fig. 1). This DSL gives an abstract and stable description of the problem to solve and also gives the possibility for the domain experts to deal with the concepts, without knowing technical details. An example of DSL is presented in Fig.1 for Product domain, used as a basic versioning system for various products, characterised by types and attributes.

After defining the domain DSL, a DSL interpreter is realized by mapping the abstract architecture on concrete components, which are called tools in our approach. The mapping is not done directly, but in terms of a feature model that captures functional and non-functional properties, which are optional and may be maintained independently from the DSL.

Once the domain and its interpreter are available, creating an application simply stands in choosing the variation points at three levels: (V1) defining the application model conforming to the DSL, (V2) selecting the features and (V3) selecting the components (tools). Thus, application development becomes a simple task that can be performed by non experts.

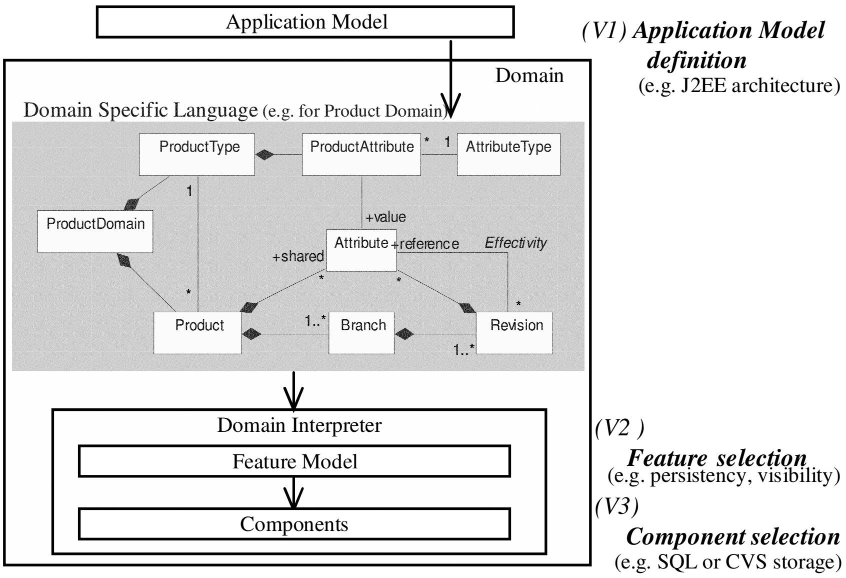


Fig. 1. Variation points inside the domain architecture

2.1.1 Application Model Definition

For a particular application, one has to define a model conforming to this DSL, which is then understood by the domain interpreter. As the model is often structural only, the application development does not need any programming [7], so it may be performed directly by domain experts and not necessarily by software engineers. E.g., if *Product* domain is used to version software artefacts conforming to J2EE architecture, a product type may correspond to the Servlet concept.

2.1.2 Feature Selection

The DSL incorporates the variability regarding the domain concepts, but there are other variations, regarding behaviour or non-functional properties, which are not related to a single concept. For this reason, the DSL, which is defined a priori, is complemented by a feature model, which may be identified and incorporated in the system a posteriori. As this feature model does not try to grasp all the variability, it is rather simple and does not need the generality of the feature models from product lines [8]. Features are optional and may be selected through the wizard given by Mélusine environment. Examples of features for *Product* domain are:

- *persistency* – for objects instantiated from `Product`, `Branch` and `Revision`;
- *visibility* – for visualizing products and revisions stored in `ProductDomain`.

2.1.3 Component Selection

The domain realization finally stands in mapping the abstract features to concrete components, which contain most of the implementation code. It has been a purpose of this approach to be able to reuse non-homogenous tools, using wrappers that adapt them to abstract components. The tools for implementing an abstract feature may be chosen through a wizard. For instance, the *persistency* feature may be mapped on components having different implementations, based either on SQL storage, or on the repository of a versioning system like Subversion or CVS.

2.2 Variation through Domain Composition

Domains are by nature narrow in scope, so applications usually span over several domains. Therefore, a domain composition mechanism is necessary, such as to:

- "match" any pair of domains, irrespective of their DSLs;
- keep unchanged the original implementation.

Being a (large) unit of reuse, a domain has an "interface", or a visible part, which is actually its DSL, expressed as classes and relationships. This DSL is used to perform composition between domains at a high level of abstraction, irrespective of features and components used at lower levels. Domain matching is performed by creating relationships between concepts, to support flexible interactions. The goal of preserving the code was obtained through aspect oriented programming (AOP).

The composition mechanism is exemplified on *RichProduct*, a composite domain reused for various purposes, from document management to software configuration management. It is obtained by composing the autonomous domains *Product* and *Document*, while preserving their original implementation. The idea behind the *RichProduct* domain was to have a versioning system for complex, "rich" elements, which are both characterized by some attributes (managed by *Product* domain) and by supplementary information stored in one or more files (managed by *Document* domain).

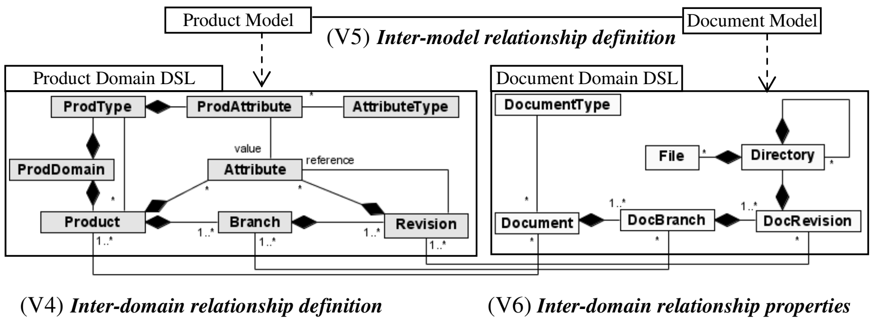


Fig.2. Variation points in the *RichProduct* composition mechanism

2.2.1 Inter-domain Relationship Definition

The composite domain DSL includes the subdomain DSLs (see an example for *Product* and *Document* composition in Fig. 2) plus relationships that either *establish inter-*

actions characteristic to the new composite domain, or link "the same" concept found in both domains - called *inter-domain relationships* (for example `Product` – `Document` or `Revision` – `DocRevision`, representing the variation point V4 from Fig.2)

Inter-domain relationships are not modifying the code of the original domains; links and interactions are implemented in a non-invasive way, using AOP.

2.2.2 Inter-model Relationship Definition

An application pertaining to a composite domain is characterized by several models, one for each of the composed domains. For developing a new application, the application designer should select a model from each domain and should define the relationships between these models, in conformity with the previously described inter-domain relationships. These inter-model relationships (variation point V5 from Fig. 2) may be established either *automatically* (if there are enough criteria for matching the model elements) or *manually*, through another Mélusine wizard (methodological details were given in [7]).

2.2.3 Inter-domain Relationship Properties

The inter-domain relationships in composite domains are characterised by several properties, which also constitute the variation points of our composition mechanism (see V6 from Fig.2). Some characteristic attributes are given below:

a) ***Destination Life Cycle Management***:
   - *Source Independent* - if objects from the source and destination classes are created independently and then related by an implicit or an explicit selection; this is similar to an association in UML;
   - *Source Dependent* - if the life cycle of the objects instantiated from the destination class is managed by the objects instantiated from the source class; these relationships are similar to UML composition but the implementation is different from a composition between classes belonging to the same model.

In our example, the documents life cycle, with branches and revisions, is under the control of products, with their respective revision tree. Thus, the destination life cycle is dependent on the source one.

b) ***Multiplicity*** – an object from the source class may be linked to one or more objects from the destination class and vice-versa; the link may be optional or not.

Multiplicity may vary with respect to the composition policy and strongly depends on the context in which the composed domains will be used. Let us take as example the multiplicity choice for the `Revision` class related to `DocRevision`. What happens when the product attributes change, while the document content remains the same? If the multiplicity is 1..*, a document revision may be linked to more product revisions.

c) ***Link Creation Moment*** - represents the moment at which the source and destination instances are linked:
   - *Early, at instantiation* - means creating the link immediately after the instantiation of the source;
   - *Late, at navigation* - means creating the link when navigation requires the destination object;

In *RichProduct,* a document is instantiated immediately after creating a product and linked to it. So happens to branches and revisions also.

d) ***Persistence*** – if the link is persistent and must be recreated in case of failure;
e) ***Captures*** - the signature of methods that have to be captured for supporting interactions across the relationship.

Mélusine environment includes a user interface that asks the composite domain developer to make the right choice for these properties. The code for implementing the above properties (like establishing and navigating the links, persistency, or destination object creation) is generated automatically in the form of AspectJ extensions. Other complex interactions between the linked objects should however be manually coded, on the basis of the skeleton generated for the captures.

3 Conclusion

The increase of size, complexity and evolution requires more flexible composition mechanisms, such as to manage variation points inside the units of reuse, but also in the composition mechanisms itself. This may be obtained through model-based composition, at a high level of abstraction. This paper presents such a composition mechanism for high granularity units of reuse called domains. Any pair of domains may be composed by establishing inter-domain relationships, whose properties represent variation points of the composition.

Acknowledgments. The work of Anca Daniela Ionita was supported by a FP6, Marie Curie – EIF fellowship at LIG-IMAG and by MEC-CNCSIS A37 grant.

References

1. van Ommering, R., Bosch, J.: Widening the Scope of Software Product Lines – From Variation to Composition. In: G. Chastek (Ed.): SPLC2 2002. LNCS 2379 (2002) 328-347
2. Favre, J.M.: Towards a Basic Theory to Model Driven Engineering. 3rd Workshop in Software Model Engineering, WiSME 2004 (2004)
3. Bruneton, E., Coupaye, T., Stefani, J.B.: Recursive and Dynamic Software Composition with Sharing. Seventh International Workshop on Component-Oriented Programming (WCOP02), Malaga, Spain (2002)
4. Peltz, D.: Web services orchestration. A review of emerging technologies, tools and standards. Hewlett Packard, Co., January (2003)
5. van Ommering, R., Beyond Product Families: Building a Product Population ?. In: F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951 (2000) 187-198
6. Estublier, J., Vega, G., Ionita, A.D.: Composing Domain-Specific Languages for Wide-Scope Software Engineering Applications. Lecture Notes in Computer Science. Proceeding of MoDELS/UML Conference, Jamaica 3713 (2005) 69 – 83.
7. Estublier, J., Ionita, A.D., Vega, G.: Relationships for Domain Reuse and Composition. Journal of Research and Practice in Information Technology, 38, 4 (2006) 287-301
8. Riebisch M., Streitferdt D., Pashov I.: Modeling Variability for Object-Oriented Product Lines. Workshop Reader of 17th European Conference on Object Oriented Programming ECOOP 2003, Darmstadt, Germany (2003) 165-178

Towards Integration of Modelling and Reusing Software Cases

Katharina Wolter[1], Lothar Hotz[1], and Thorsten Krebs[1]

HITeC e.V. c/o Department Informatik, Universität Hamburg
Hamburg, Germany, 22527
{hotz|krebs|kwolter}@informatik.uni-hamburg.de

Abstract. This paper introduces a novel approach to integrate reuse of software cases and dynamic variability modelling. Software cases comprise both a problem description in form of a requirements specification and a solution in form of architecture, design and code. Previous software cases are identified based on the requirements specification and can be reused for different but similar problems. This paper describes work in progress. The ideas stem from research work within the EU-funded project ReDSeeDS (Requirements-driven Software Development)[1].

1 Introduction

The necessity to enhance reuse in software development is known. Software product lines and variability modelling are well-known approaches to reach this goal [1] [2]. Adopting these approaches, the variability of the domain is modelled *in advance* to software development and is reused during application engineering.

In this paper, we propose a novel approach in which reuse is based on software cases and variability can be modelled *during* software development. Using this approach, the additional effort for variability modelling is kept minimal. Our approach clearly differs from the software product line approach. New variability and new artefacts are defined during application engineering, when needed. The approach can be seen as a complement or alternative to the strict separation of application engineering and domain engineering which is typical for development in software product lines.

A *software case* comprises a problem statement (requirements) and a solution (architecture, design and implementation) [3]. The requirements specification is mapped to appropriate elements of the solution. Between the solution elements, there is also a mapping defined: i.e. between requirements and architecture, between architecture and design, and between design and code. For a more detailed description of this approach we refer the interested reader to [3].

All software cases of one domain are collected in a *software knowledge model*. While a single software case does not contain any variability, each new software case introduces new variability. The variability in the software knowledge model

[1] http://www.redseeds.eu

is exploited during software development as follows. Based on a partial requirements specification *similar* software cases are identified [3]. From these retrieved software cases, both requirements and the corresponding solutions can be reused. The resulting new software case is added to the software knowledge model.

In order to compute similarity between different software cases it is apparent that the software knowledge model must be consistent at any time. A challenge to this is the fact that customers use their own, probably domain-specific, terminology. Hence, there is no explicit one-to-one mapping from the requirements of a new software case to requirements of stored software cases. For keeping requirements customer-understandable, but being able to map their requirements to stored cases, a *local vocabulary* is created for every software case. Terms of this vocabulary contain a mapping to terms defined in a global, common vocabulary which is maintained within the software knowledge model. The mapping between the local vocabulary and the common vocabulary can be one-to-one, but also include synonym and homonym definitions. Whenever requirements specifications contain new terms that the global vocabulary does not yet contain or terms used with a different meaning, these are integrated into the common vocabulary – henceforth called *consolidated vocabulary.* This approach allows to define different, customer-related terminologies but nonetheless using the full power of similarity-based identification and reuse of different software cases. The software knowledge model grows incrementally and thus can become very large. In order to ensure consistency, tool support is essential.

The remainder of the paper is organised as follows. First, we illustrate how software development looks like using a software knowledge model for defining variability of stored software cases (2.1). Then we detail the software knowledge model (2.2) and show how similarity of software cases can be computed based on the consolidated vocabulary (2.3). We discuss and compare the approach to related work (3), present future work and finally summarise the main benefits (4). Throughout the paper we use examples from the domain of management software applied for a fitness club.

2 Case-base variability modelling

2.1 Vision

Within the ReDSeeDS approach [4, 3], reuse already starts during requirements specification. The Requirements Engineer (RE) defines requirements e.g. in a set of scenarios[2]. Each scenario is a sequence of sentences written in restricted English[3], called *requirements statements.* A tool supports the RE during requirements specification. When the RE enters a new sentence the tool checks

[2] Note that the ReDSeeDS approach also supports other forms of requirements specifications. Fore simplicity reasons we restrict ourselves to scenarios in this paper.

[3] One example of restricted English is *Subject-verb-object (SVO)* sentences where the subject comes first, the verb second and the object third [5]. There are also other approaches to restrict English text, like the *Attempto Controlled English (ACE).* The extended form of SVO sentences used in ReDSeeDS have the advantage of

the consolidated vocabulary for terms of the statement and displays relevant parts, if any. The following cases need to be distinguished:

1. *The term is already defined in the software knowledge model*
   (a) *Both terms have the same meaning*
   The term can be reused and the software knowledge model does not need to be changed.
   (b) *The new meaning differs from the one in the software knowledge model*
   An example for this are *homonyms*. The term 'enter' for example can describe that a person goes into a room or that a person types in data. The software knowledge model needs to be extended with the new meaning.
2. *The term is not yet defined in the software knowledge model*
   (a) *The new meaning is already defined using a different term*
   An example for this are *synonyms*. The term 'enter' may already be defined but 'typing' is not. The software knowledge model needs to be extended with the new term.
   (b) *The new meaning is not yet defined*
   The software knowledge model needs to be extended with the new term and its meaning.

Please note that the RE usually will not capture all requirements from scratch. One main benefit of the ReDSeeDS approach is that the RE specifies some initial requirements and then uses a query engine to search for software cases with similar requirements [3]. Because of the traceability links between requirements, architecture, design and code it is possible to reuse appropriate solutions to the problem specifications. This means that the RE can reuse parts of problem descriptions together with the associated parts of the problem solutions from several former software cases. These partial problem descriptions together with associated problem solutions are called *subcases* in the following.

The consolidated vocabulary enables measuring of similarity of requirements even without lexical correspondence between the statements. Although the consolidated vocabulary can be very large, integration of new terms and maintenance are supported by structuring the terms in an ontology (see next section).

The software development process originally performed in an organisation needs to be adapted to this approach. However, there are only slight differences compared to the implementation of a traditional product line approach. A main difference is in the set of activities to be performed in software development: besides traditional development activities (i.e. making architecture and design decisions, coding, etc.) new activities are related to the integration of subcases from different software cases and their adaptation. Methods like generic components or aspect-oriented programming can support this task. Comparing new customer-specific terms to existing terms in the consolidated vocabulary, and, if needed, extending this vocabulary at appropriate places are new activities, also.

being syntactically unambiguous in contrast to natural language which can also be used in ReDSeeDS if necessary.

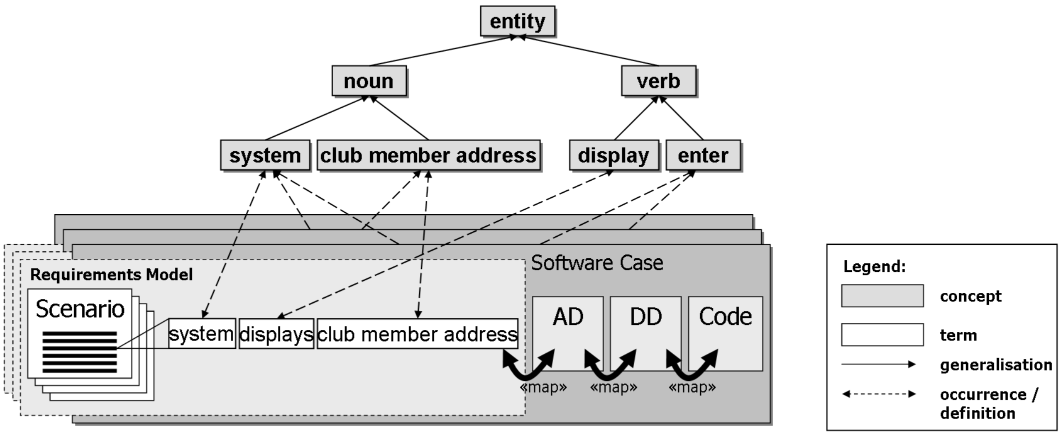


Fig. 1. The software knowledge model.

2.2 Software Knowledge Model

The *software knowledge model* contains a representation for complete software cases (requirements, architecture, detailed design, and code) and the consolidated vocabulary. Since we present a first step towards this uniform representation in this paper, we limit ourselves to modelling requirements.

We propose to use a generic logical representation for the software knowledge model and thus for representing variability (see for example [6]). *Concepts* are used to represent requirements, components, code, and other artefacts used for software development. A *taxonomy* defines an inheritance structure between concepts and a *partonomy* composes concepts by relating aggregates and parts. Figure 1 depicts a taxonomy of terms that are modelled by means of concepts. These concepts are related to the terms in all software cases they are used in, as specified with the *occurrence* relation (*definition* relation in the other direction). Structuring the vocabulary in a taxonomy has some key benefits:

– The system's knowledge of a term is defined by the concept representing the term and its relations (specialisation, composition, etc.). As a result semantic-based similarity measures can be applied (see next section).
– Homonyms can be disambiguated by the specialization paths of the terms. The two meanings for the term 'enters' for example can be identified by the path via 'moving' for 'club member enters fitness club' and 'typing' for 'staff member enters club member address'.
– The vocabulary can be browsed for identifying reusable terms.

2.3 Software Case Retrieval

For identifying cases, (partial) requirements of the current software case are compared to requirements of stored cases. With the consolidated vocabulary, the terms of all software case are comparable: the ontological structure can be exploited to identify similarity of terms from different cases.

A child concept is similar to its parent because the former is a specialization of the latter. Two siblings are also similar because both are children of the same parent (they inherit the same characteristics). Concepts in different places of the taxonomy are not necessarily similar. For example 'staff member' and 'club member' are both 'person's which means that they are siblings and have a lot in common (e.g. name, address, etc.) compared to a 'staff member' and a 'bracelet' which is a 'thing'. Several heuristic similarity measures that are based on a common hierarchial structure of terms are described in [7].

3 Discussion and Related Work

In case-based reasoning, *cases* (i.e. past experience) are stored to be retrieved for similar problems [8]. The solution in the retrieved case is examined and applied to the current problem with suitable modifications. While most case-based approaches to software reuse apply reuse on the level of software components and code [9], ReDSeeDS starts reuse already on the level of requirements. This is an essential advantage due to the fact that former cases can be identified early and requirements specification can be skipped when requirements are reused together with the solution.

Software Product Lines (SPLs) are a means for large-scale reuse. A product line contains a set of products that share a common, managed set of features satisfying the specific needs of a particular market segment or mission [1]. Domain engineering plays a key role: analysing the product domain, building a common architecture and planning reuse. Variability is modelled explicitly e.g. in a feature model. This modelling can be done in a waterfall-like approach or incrementally. Our approach models variability is implicitly through the different software cases. In contrast to a-priori domain engineering, our approach enables dynamic extension of the reusable asset repository, i.e. reuse is not planned a-priori. Evolving the model during product development contains both, evolving the problem description (vocabulary, scenarios, etc.) and including the solution to the problem (i.e. architecture, design and code). This means that domain engineering is not a separate tasks but done during application development. In SPL reusable artefacts are determined using a variability model while in the ReDSeeDS approach these artefacts are identified by searching for software cases with similar requirements.

4 Summary and Future Work

In this paper, we have introduced a novel approach that integrates variability modelling and reuse of former software cases. A software case comprises the problem description in form of a requirements specification and a solution description in form of architecture, detailed design and code artefacts. Reusing software cases or subcases is enabled with a query engine that compares requirements specifications of former cases to the current problem. Variability is modelled implicitly by the set of all known software cases and is automatically extended for every new software case that is developed and stored.

The ideas in this paper originate from research work within the ReDSeeDS project, in which a similarity measure will be developed based on the consolidated vocabulary sketched in this paper. The AMPL (Asset Modelling for Product Lines) language developed in the ConIPF (Configuration in Industrial Product Families)[4] project defines a language to represent complete product knowledge (incl. features, context, software and hardware artefacts) [6]. These modelling facilities will be extended to formalize requirements, architecture, design and code. The adaptation of subcases from former software cases and the integration of subcases from different software cases are further challenges.

Acknowledgement

This work is partially funded by the EU: Requirements-driven Software Development System (ReDSeeDS) (contract no. IST-2006-33596). The project is coordinated by Infovide, Poland with technical lead of Wasaw University of Technology and with University of Koblenz-Landau, Vienna University of Technology, Fraunhofer IESE, University of Latvia, HITeC e.V. c/o University of Hamburg, Heriot-Watt University, PRO DV, Cybersoft and Algoritmu Sistemos.

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
2. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021 (1990)
3. Smialek, M.: Towards a Requirements driven Software Development System. In: Models 2006. (2006)
4. Smialek, M.: Acoomodating Informality with Necessary Precision in Use Case Senarios. Journal of Object Technology **4**(8) (2005) 59–67
5. Graham, I.: Task scripts, use cases and scenarios in object oriented analysis. Object Oriented Systems **3** (1996) 123–142
6. Hotz, L., Wolter, K., Krebs, T., Deelstra, S., Sinnema, M., Nijhuis, J., MacGregor, J.: Configuration in Industrial Product Families - The ConIPF Methodology. IOS Press, Berlin (2006)
7. Pedersen, T., Patwardhan, S., Michelizzi, J.: Wordnet::similarity - measuring the relatedness of concepts. In: In Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04). (2004)
8. Aamodt, A., Plaza, E.: Case-based reasoning: Foundational issues, methodological variations, and system approaches. Artificial Intelligence Communications **7**(1) (1994) 39–59
9. Fouque, G., Matwin, S.: A case-based approach to software reuse. Journal of Intelligent Information Systems **2**(2) (1993) 165–197

[4] http://www.conipf.org

Goal and Variability Modeling for Service-oriented Systems: Integrating *i** with Decision Models

P. Grünbacher[1] D. Dhungana[1] N. Seyff[2] M. Quintus[2] R. Clotet[3] X. Franch[3] L. López[3] J. Marco[3]

Christian Doppler Lab for Automated Software Eng.[1]
Johannes Kepler University
4040 Linz, Austria

Systems Engineering and Automation[2]
Johannes Kepler University
4040 Linz, Austria

Universitat Politècnica de Catalunya (UPC)[3]
Barcelona, Spain

Paul.Gruenbacher@jku.at

Abstract. Variability modeling and service-orientation are important approaches that address both the flexibility and adaptability required by stakeholders of today's software systems. Goal-oriented approaches for modeling service-oriented systems and their variability in an integrated manner are needed to address the needs of heterogeneous stakeholders and to develop and evolve these systems. In this paper we propose an approach that complements the *i** modeling framework with decision models from orthogonal variability modeling. We illustrate the approach using an example and present options for tool support.

1 Introduction

Stakeholders of today's software-intensive systems demand flexibility and adaptability to allow rapid system evolution made necessary by new and changing requirements. Variability modeling and service-orientation are promising with respect to both flexibility and adaptability. Variability modeling is an approach fostering software reuse and rapid customization of systems [8][10]. Service-orientation is visible in buzzwords such as service-oriented computing, service-oriented architectures, or service-oriented software engineering and is promoted by a number of emerging standards for service-oriented development. Recently, researchers have started to explore the integration of service-oriented systems and variability modeling. Variability modeling is increasingly seen as a mechanism to support run-time evolution and dynamism in different domains and to design, analyze, monitor, and adapt service-oriented systems [7]. At the same time the modeling framework *i** [11] is gaining popularity to model service-oriented and agent-based systems [9] and researchers are seeking new ways to enhance it with variability modeling capabilities [6].

Pursuing similar goals we have been using *i** to model a service-oriented multi-stakeholder distributed system (MSDS) in the travel domain to validate its usefulness in this context [3]. Despite the power and expressiveness of *i** we experienced some deficiencies when modeling variability in particular when specifying the needs of heterogeneous stakeholders in the MSDS or when investigating the modeling needs of service provides and service integrators. As a result we started investigating the dependencies of goal modeling and variability modeling. In this paper we discuss an initial approach integrating orthogonal variability modeling techniques into *i**. We illustrate the approach using examples and discuss tentative tool support based on our existing work on meta-tools for variability modeling. Our approach is based on our integration framework [3] as well as our earlier work on the use of *i** [5] and variability modeling [4].

2 Modeling the Variability of Service-oriented Systems with *i**

Modeling service-oriented systems requires an understanding of stakeholder goals and goal variability. *i** is an established framework for goal modeling [11] which is goal-oriented as well as actor-oriented and supports the assignment of responsibilities to system actors to express high-level actor requirements. There are two types of models in *i**: Strategic Dependency (SD) models define actors, their relationships (e.g., specialization and composition) and how they depend on each other. Strategic Rationale (SR) models state the main goals of these actors and their decomposition using some refinement constructs. Together the SD and SR models provide a comprehensive system overview. *i** supports recording the rationale behind requirements and decomposing elicited requirements at the required level of detail. At the requirements level actors are mainly used to represent stakeholders' needs, while at the architecture levels they can be used to model services: For instance, Franch and Maiden have explored the use of *i** to model architectures using roles and agents [5]. A similar approach has been proposed by Penserini *et al.* in [9]. *i** supports traceability from high-level actor goals to concrete services in the running system and vice versa. It has been shown that high-level stakeholder goals tend to be more stable than underlying requirements and selected software solutions. Linking services to high-level stakeholder goals modeled in *i** thus increases system stability and adaptability by guiding the replacement of malfunctioning services with services also fulfilling essential stakeholder goals. It also facilitates the identification of affected stakeholders [3].

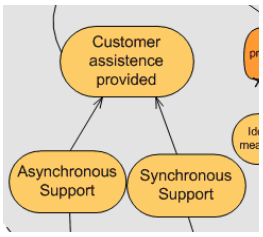


Fig 1. Modeling Goal Variability in *i**: Customer assistance can be either provided using asynchronous or synchronous support.

We are currently exploring the benefits and limitations of *i** for developing and evolving service-oriented systems. A key experience is that variability essential for modeling service-oriented systems at different levels of abstraction. Figures 1 and 2 show partial *i** examples of external and internal variability.

Our framework presented in [3] defines four different modeling layers for service-oriented systems: stakeholder needs, architecture prescription, solution architecture, and open system. Figure 2 shows a concrete modeling example on the architecture layers. For instance, the architecture prescription layer may define the actor "Travel services provider" (expressed as role in *i**) for undertaking the "Book hotel" and "Book flight" system tasks. At the lower solution architecture layer several services cover the role "Travel services provider": The services "Amadeus" and "Schubert" are modeled as *i** agents since they are real-world entities. The open system instance layer describes a running system. If the service "Amadeus" is chosen one may choose the "Spanish Amadeus Server" service hosted on a Spanish site or the "Austrian Amadeus Server" hosted on an Austrian site. Again these services are modeled as *i** agents, related to "Amadeus" by using the *instance* relationship in *i**.

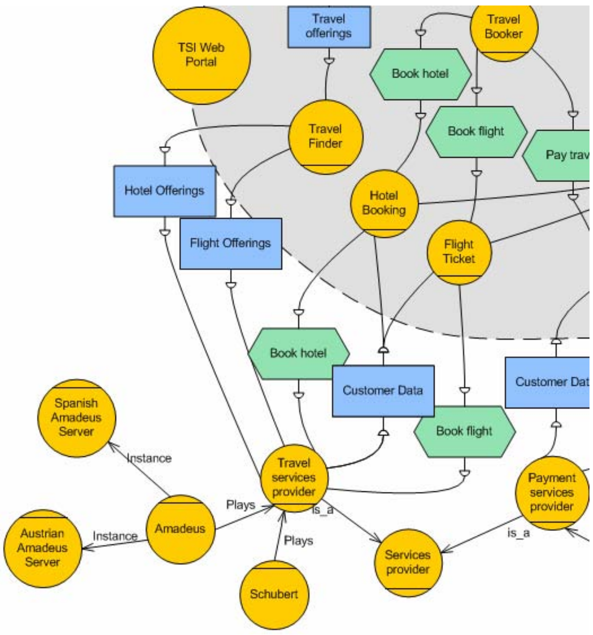


Fig 2. Modelling Service Types and Services in *i**.

The example shows the capabilities of *i** for modeling service-oriented systems at different levels of abstraction. The language can be used to model high-level concerns such as stakeholder goals, architecture-level aspects, and even the configuration of the open system based on service instances [3]. The flexibility of *i** was also pointed out by other authors [1]. Traceability is a major benefit: The *contributes* relationship in *i** allows establishing traceability between stakeholder goals, service types, selected services, and service instances.

The examples, however, also show some limitations of modeling variability in *i**. The expressiveness and formality is insufficient compared to existing variability modeling approaches. There are no language constructs to capture more formal aspects required in variability models such as constraints (e.g., between services), conditions under which services become active or inactive, selector types, or cardinalities [10]. The variability modeling capabilities of *i** should therefore be enhanced.

3 Using Variability Modeling with *i**

We propose an approach based on our framework for multi-stakeholder distributed systems [3] and our earlier work on the use of *i** [5] and meta-tools for variability modeling [4]. A fundamental approach in variability modeling is to complement existing models and artifacts with variability information rather than using specific notations or languages. Our work is influenced by a decision-oriented approach proposed by Schmid and John [10] that supports orthogonal variability modeling for arbitrary artifacts independent from a specific notation. The benefits of such approaches are the flexibility gained and traceability established by using one variability mechanisms for different artifacts at the requirements, design, architecture, implementation, application, and runtime level. We propose to complement *i** with orthogonal variability modeling techniques. Such an approach requires:

- The development of a *decision model* describing the variability of the system and dependencies between variabilities [4].
- An *asset model* describing the system elements and their dependencies [4]. In the domain of service-oriented systems the elements include service types, services, and service instances together with their dependencies (e.g. a payment service might rely on a transaction service).
- The annotation of *i** models with rules referring to the decision model to model inclusion conditions for services and the dependencies among assets and decisions.
- A mechanism to prune *i** models based decisions taken at design-time or runtime to generate/update system configurations on the fly (e.g., by adding/removing/updating services).

We envisage an *i** model to hold a snapshot of a service-oriented system at a certain point in time that can be adapted based on decisions taken by stakeholders by considering all assets and their dependencies. In the product line terminology such a snapshot is based on the domain-level, ”product-line“ version of the *i** model. Obviously, beyond the *i** model, the runtime configuration requires the generation of additional information for configuration, i.e., concrete values of decision variables that inform system configuration (see Figure 3).

Decision Variable	*Question*	*Selection Type*	*Cardi-nality*	*Link to i* element*
Type of customer assistance	What kind of customer assistance do you need?	Set {Synchronous Support, Asynchro-nous Support}	1:2	Customer assistance provided
Degree of customer assistance	How many hours per day should the hotline be available?	Value [0..24]	1	Customer assistance provided
Travel Service Provider	Which is your preferred travel service?	Set (Ama-deus, Schu-bert)	1	Travel Service Provider

Fig 3. Partial decision model.

4 Adopting a Variability Modeling Meta-Tool

We are aiming to provide tool support for the discussed approach and have been tailoring the orthogonal variability modelling meta-tool DecisionKing to our problem context [4]. DecisionKing allows the definition of meta models for arbitrary asset types to create a customized variability modeling tool. The tailoring of a custom-built variability modeling tool in DecisionKing covers (i) the definition of a domain-specific meta-model and (ii) the development of domain-specific plug-ins:

Definition of the meta-model for service-oriented variability modeling. This step covers the identification of the relevant asset and dependency types. We identified the asset types goal, service type, service, and service instances: A *goal* of a stakeholders maps to an actor goal in *i**. Different *services types* contribute to fulfilling these goals (e.g., "Travel services provider"). Available services realizing a service type are modeled as a *service* (e.g., "Amadeus"). Finally, available runtime implementations of services can be modeled as service instances (e.g., "Spanish Amadeus Server"). We also identified two kinds of relationships between the assets: The *requires* relationship is used whenever the selection of a certain assets leads to the selection of another asset. This can be a result of logical dependencies between goals, conceptual relationships between service types, relationships between services, or functional dependencies between service instances. The *contributesTo* relationship is used to capture structural dependencies between assets of different levels. Service instances for example contribute to services. Services contribute to service types which contribute to goals. It is however also possible that goals are split up into sub-goals. Such compositional relationships between goals can also be modeled using the *contributesTo* relationship.

Development of plugins. DecisionKing's capabilities can be extended by plugging-in domain-specific functionality [4]. Using this mechanism we are developing a link between DecisionKing and the *i** modeling tools REDEPEND using an XML-based interchange language for our tool suite.

5 Open Issues

In this paper we proposed an initial approach to complement *i** with an orthogonal variability modeling technique. There are several open issues needing attention:

We need to extend the *i** language in order to include variability information. On the one hand, we need to provide complete formal semantics for the *is_a* inheritance *i** mechanism, which is currently only defined at the actor level. On the other hand, we need to provide a formal syntax for modeling taken decisions (e.g., which services are chosen) and variation points. We are considering the use of the *i** routine concept for reflection decisions taken in the model.

We also need to complete tool integration to improve traceability between *i** and variability models. For this purpose we will link the *i** meta-model and the variability meta-model and exchange information between models using an XML interchange definition language currently under definition.

References

[1] J. Castro, M. Kolp, J. Mylopoulos. "Towards Requirements-Driven Information Systems Engineering: The Tropos Project". *Information Systems*, vol. 27, 2002.

[2] L. Chung, B.A. Nixon, E. Yu, J. Mylopoulos, Non-functional requirements in software engineering, Kluwer Academic Publishers, 2000.

[3] R. Clotet, X. Franch, P. Grünbacher, L. López, J. Marco, M. Quintus, N. Seyff: "Requirements Modelling for Multi-Stakeholder Distributed Systems: Challenges and Techniques". RCIS'07: 1st IEEE Int. Conf. on Research Challenges in IS, Ouarzazate, 2007

[4] D. Dhungana, P. Grünbacher, and R. Rabiser, "DecisionKing: A Flexible and Extensible Tool for Integrated Variability Modeling.", 1st International Workshop on Variability Modelling of Software-intensive Systems, Limerick, Ireland, 2007.

[5] X. Franch, N.A.M. Maiden. "Modeling Component Dependencies to Inform their Selection". In *Proceedings 2nd International Conference on COTS-Based Software Systems* (ICCBSS), Lecture Notes on Computer Science 2580, Springer,2003.

[6] S. Liaskos, Y. Yu, E. Yu, J. Mylopoulos. "On Goal-based Variability Acquisition and Analysis". *Proc. 14th IEEE Int'l Requirements Engineering Conference* (RE'06) (Sept 11-15, 2006). IEEE Computer Society

[7] J. Peña, M.G. Hinchey, A. Ruiz-Cortés. "Multi-agent system product lines: challenges and benefits". *Communications of the ACM*, vol. 49, n. 12 (Dec. 2006), 82-84.

[8] K. Pohl, G. Böckle, and F. J. van der Linden, Software Product Line Engineering: Foundations, Principles, and Techniques: Springer, 2005.

[9] L. Penserini, A. Perini, A. Susi, J. Mylopoulos. "From Stakeholder Needs to Service Requirements". *Proceedings of the 2nd International Workshop on Service-Oriented Computing: Challenges on Engineering Requirements* (SOCCER), 2006.

[10] 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, vol. 53(3), pp. 259-284, 2004.

[11] E. Yu. Modeling *Strategic Relationships for Process Reengineering*, PhD Thesis, Toronto, 1995.