Variability Models Must Not be Invariant!

Elmar Juergens and Markus Pizka
Technische Universität München
Institut für Informatik
{juergens, pizka}@in.tum.de

Abstract

Variability modeling techniques are used to specify variable aspects of members of a family of related software artifacts. Instances of variability models are then used to efficiently produce members of such a family. By making variability explicit, variability models determine implicitly the common properties among family members as well.

This partitioning of information into variable and invariant parts predetermines the reuse benefit obtainable from a variability model. In most current approaches to variability modeling, the decision between variable and invariant information has to be done in an up-front manner and is very difficult to change later on. However, the distinction between variable and common parts of members of a system family varies over time. Variability modeling techniques must thus be able to cope with changes to the variability models.

Since many variability modeling techniques do not currently support this, research is required to allow an evolution of the distinction between what is variable and what is invariant over time. This paper elaborates on the necessity, related work and possible approaches to tackle this challenge.

Keywords *modeling, software evolution, formal methods, software reuse*

1 Introduction

Since the term *Software Engineering* was coined in 1968 [14], software reuse is considered a potential means to alleviate the difficulties encountered in the construction of large software systems. It promises to decrease costs and increase quality by reusing existing components for the construction of new systems. This hope also drives several approaches that receive increased attention in recent years, such as *generative programming*, *model driven software development* and *software product lines*.

As accurately noted by Krueger in his influential paper on software reuse [9], abstractions are the basis of all reuse mechanisms. In [17], Wegner illustrates the close relationship between abstractions and reuse by pointing out that every abstraction describes a collection of related, reusable entities, and that vice versa, every collection of related, reusable entities determines an abstraction. Hence, in order to better understand the potential of approaches to software reuse — both old and new ones — it is important to understand the principles of abstraction.

In order to better understand the benefit of variability modeling approaches (such as *model driven software development* and *software product lines*), we investigate in this paper various abstraction mechanisms from the perspective of software reuse. This investigation reveals that the partitioning of information into variable and invariant parts of an abstraction is the crucial factor determining the overall benefit of an abstraction for software reuse. This paper further shows that it is impossible to achieve an optimal partitioning in practice, if it cannot be altered during the lifetime of an abstraction, since the creation of an optimal partitioning would requires complete and in-advance knowledge of future requirements. From this, we conclude that in order to exploit the optimal reuse benefit of an abstraction, it is of utmost importance to be able to evolve the partitioning into variable and invariable properties itself over the lifetime of the abstraction.

Outline
The remainder of this paper is structured as follows:

Section 2 investigates the role of variability in different abstraction mechanisms from the perspective of reuse.

Section 3 illustrates the impact that the amount of invariant information has on the reuse benefit an abstraction provides.

Section 4 gives additional evidence for the claim that increased variability is needed by referencing numerous iso-

lated approaches to tackle this problem for different abstraction mechanisms.

Section 5 concludes the paper with the appeal to join research efforts in order to move from isolated ad-hoc approaches to a general, unified understanding of variability evolution in abstraction mechanisms.

2 Abstractions, Reuse and Variability

According to Krueger [9], every abstraction mechanism employed in software development comprises two levels: the higher referred to as *specification* and the lower as *realization* of an abstraction. Specifications define the variability among the realizations of an abstraction. Different instantiations of the variability defined within an abstraction specification map to different realizations of the abstraction. This perception of abstractions is depicted graphically in Figure 1. The variability exposed by the abstraction specification is depicted by red, blue and green triangles. Depending on the variable information, the abstraction specification maps to different abstraction realizations. This mapping is performed by abstraction mechanism-specific translation mechanisms. (In the case of programming languages for example, it is performed by the compiler, for domain specific languages by the generator and for libraries by the developer that implements the library's interfaces and methods.) The colored half-circles depict those parts of the abstraction realizations that are generated from the variable information. The commonality among the realizations is depicted by gray squares.

By making variability explicit, a specification also implicitly defines the invariant parts of an abstraction. Since they are common among all realizations of an abstraction, they hold the potential for reuse: the bigger the invariant parts are, the more information can be reused between abstraction realizations.[1]

This consideration of abstractions as specification and realization is still as applicable to the conception of abstraction mechanisms today, as it was in 1992. Let us consider various commonly used reuse mechanism from this perspective.[2]

Assembly Language: Every statement in assembly language abstracts from a micro program in machine code. Assembly language statements are abstraction specifications, the micro programs they are translated to are the corresponding abstraction realizations. The variability consists of the parameters offered by the

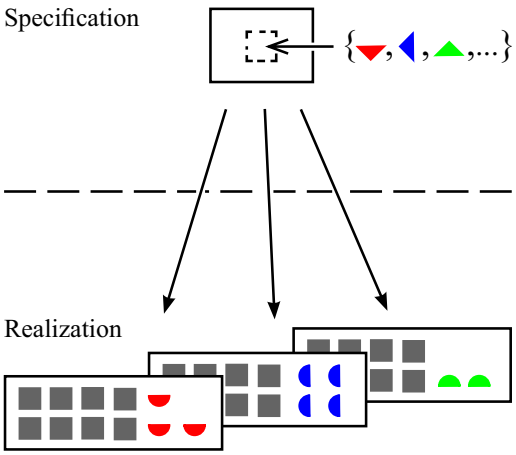

Figure 1. Abstraction Principles

statements. Every aspect of a statement that cannot be parameterized is common among all corresponding machine code programs.
The statement `MULT <reg1>,<reg2>` for example performs the multiplication of 32 bit numbers. The registers from which the numbers are read, are variable. However, the actual algorithm used for multiplication, or the register in which the result is stored, is invariant and thus common among all micro programs generated from `MULT` statements.

High Level Programming Languages (i.e. C/C++, Java): Every construct in a high level programming language specifies an abstraction that is realized by an assembly language program. Again, the variability is defined by the parameters provided on the specification level.
The statement `while(<expr>) {<block>}` for example loops over the statements in `<block>`, as long as `<expr>` evaluates to true. Parameters `<expr>` and `<block>` are variable, the fact that a loop is performed, and the assembly language statements generated for it are invariant, however.

Libraries: High level programming languages provide abstraction mechanisms (i.e. methods, classes) that are employed to create abstractions that can be reused in programs in that language. Libraries are collections of such abstractions. The public interface provided by a class in a library corresponds to the specification, its implementation to the realization of an abstraction.
`MsgBox("Low Battery", vbWarning)`[3] for example opens a dialog that displays a warning message. The parameters of the method make the warning

[1] Krueger further divides the invariant information into a visible and a hidden part. The hidden part comprises those common aspects among abstraction realizations, which are not "visible" in the abstraction specification. For the sake of simplicity, we omit the distinction between visible and hidden aspects of invariant information in abstractions in this paper.

[2] Several of these observations can already be found in [9]

[3] In Microsoft Visual Basic 6.0

and the message type variable. All other aspects — such as color, widgets, displayed icons, size or layout of the dialog — are invariant.

Domain Specific Languages (DSLs): DSLs allow the concise expression of problems from a specific technical or business domain by offering abstractions from that domain as first-class language elements. These language elements are abstraction specifications that are often translated to programs in high level programming languages.
An example for this are regular expressions, from which Lex[4] generates finite state machines that perform tokenization in compiler front ends. In this case, the regular expressions define the variability: Lex can generate finite state machines for all regular languages. However, the finite state machine implementation, the target language and the input text encoding are invariant and thus common among all generated lexers.

Software Product Lines: In software product lines, members of a family of software systems are created by composition of standardized components. In feature modeling tools such as *fmp*[5], configurations serve as composition descriptions. In such a setting, feature diagrams serve as abstraction specifications. The systems that are composed from their configurations correspond to the abstraction realizations.
In software product lines it is especially obvious that any aspect that is not explicitly made variable in the feature diagram is invariant and thus necessarily common among all realizations.

The above mentioned abstraction mechanisms are often layered on top of each other in practice: Micro programs are generated from assembly language, assembly language is compiled from programs in high level languages, and so on. Every layer increases the level of abstraction of the highest abstraction specification.

For software developers working on the highest level of abstraction in this stack has two major consequences:

Increase in productivity: developer productivity increases, since many details are "abstracted away" and taken care of by the translation to lower levels. For example, letting the compiler take care of register allocation saves developer time and effort. Regarded from the perspective of variability and invariance in abstraction specifications, this means moving register allocation from the variable to the invariant part: the algorithm employed for register allocation is invariant for all high level language programs that are translated by a compiler.

Decrease in applicability: The developer loses some control over the final result. E. g. since the compiler takes care of register allocation, a developer cannot influence it anymore, even if it was needed. In this example, this limits the applicability of higher level programming languages for system level programming.
This negative effect necessarily increases with the level of abstraction. The more invariant information an abstraction contains, the more specialized and thus the less widely applicable it becomes. Obviously, a DSL for parser generation cannot be used to program a web server.

Making information in an abstraction invariant thus increases productivity and decreases applicability. This gives rise to the crucial question of abstraction creation:
What should be variable, and what should be invariant?

We are convinced that (in general), this question cannot be answered satisfactorily once for the entire lifetime of an abstraction. Instead, we believe that it must evolve continuously in order to reflect unanticipated changes.

3. Invariance Dilemma

The overall reuse benefit that the creation of an abstraction brings to software development, depends on at least two things: how much effort does the application of the abstraction save, and how many times can the abstraction be applied.

Unfortunately, both factors depend on the level of abstraction of the abstraction specification, and thus on its amount of invariant information. Hence, increasing the level of abstraction reduces the applicability, and vice versa, increasing the applicability by reducing invariant information reduces the level of abstraction:

- Increasing invariant information increases the amount of knowledge that can be reused among abstraction realizations. The more invariant information, the higher thus the saving in developer effort every time the abstraction is used.
  However, more invariant information requires more commonality among the abstraction realizations and thus reduces the number of opportunities for applying the abstraction.
- Reducing invariant information reduces the distance between specification and realization and thus the size of the reused artifacts. Reducing invariant information thus reduces the amount of saved development effort for each use of an abstraction.

[4] The Lex & Yacc Page: http://dinosaur.compilertools.net
[5] Feature Modeling Plug-in: http://gp.uwaterloo.ca/fmp

However less invariant information relaxes the commonality requirement — an abstraction can be applied more often.

The optimal amount of invariance in an abstraction can thus only be determined with respect to the abstraction's use cases: ideally, all common aspects of the use cases of an abstraction should be made invariant. In other words: if the use cases of an abstraction are not known, it is not possible to determine the optimal partitioning of information into variable and invariant parts of an abstraction. Since in practice the use cases of an abstraction (including future use cases!) are hardly ever known, we call this observation the *invariance dilemma*.

Let us consider the development of a product line as an example to make this dilemma more tangible. The Feature Diagram in a product line defines the variability (and commonality) among products. Every single feature that is added to the diagram decreases the level of abstraction, but increases the size of the product family (and thus the potential number of applications).

If abstraction specifications cannot be altered during their lifetime, abstraction creation is thus the quest for the lesser evil: waste of potential productivity increase or waste of potential abstraction applications.

This dilemma can be elegantly avoided, if abstractions can be easily evolved during their lifetime. We start by creating an abstraction specification that only considers variability among the already known use cases. All commonality is made invariant. As soon as a new use case comes up that requires to make a certain — formerly invariant — part variable, the abstraction is adapted accordingly. This way, abstractions start at the highest possible level of abstraction. Every decrease of abstraction is however justified by an additional application, avoiding the decrease in overall reuse benefit.

If this procedure is followed consequently, this bottom-up way of abstraction creation always achieves optimal reuse benefit. However, in order to exploit this in practice, it must be possible to adapt the separation between variable and invariant information during the life time of the abstraction. In other words: **variability models must be variable!**

4. Variability Evolution

Since abstractions play such a central role in software engineering, changes to an abstraction can affect many artifacts in a development process. In general, changes to the specification of an abstraction affect the already existing instances of the abstraction and the tools used for processing them.

Figure 2 illustrates this problem, using Domain Specific Languages as exemplary abstraction mechanism. Every change to the specification of a language requires compensating migrations of the words[6] and adaptations of the processing tools[7] of the language.

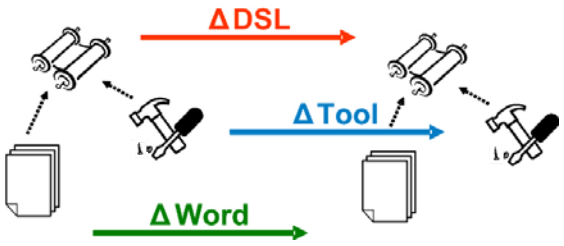

Figure 2. Abstraction evolution triggers migration of existing words and adaptation of processing tools

Performing these migrations manually is tedious, error prone and costly. Abstraction mechanisms that do not automate these compensational efforts effectively inhibit abstraction evolution in practice.

Partial solutions to automate the compensational effort triggered by abstraction evolution have been developed for various abstraction mechanisms. The existence of these efforts substantiates our claim for the need of abstraction mechanisms that support variability evolution in practice. These approaches include:

Schema Evolution: The schema of a database system [8] determines which entities and relationships — and attributes thereof — can be stored in a database. All aspects not contained in the schema are not considered. The decision of which information to include into a schema, is related to the decision which information to make variable in an abstraction. Schema evolution [2, 3] deals with altering this decision — that is the entities, relations and attributes of a schema — while the schema is in use.

Schema evolution research mainly deals with the problem of migrating instance data. Adaptation of tools is only, if at all, considered laterally. However, after decades of mainly academic interest, some transition to industrial projects seems to be taking place [1].

Grammar Engineering: Grammars of different forms[9] are used to describe the syntax of formal languages

[6] The term "word" is used as in formal language theory to depict strings that conform to the language syntax.

[7] I.e. compiler, debugger, pretty printer, ...

[8] Independent of whether it is relational, hierarchical or object oriented

[9] I.e. context free grammars, XML schemas or document type definitions. Compare [8] for a more comprehensive list.

and the structure of software systems. Grammar engineering [8, 10] deals with the systematic development and maintenance of grammars. In [12, 11], Lämmel also investigates the coupled evolution of grammars and their words.
While these approaches constitute considerable progress in the field, a lot of research is still necessary in order to better understand and master the evolution of grammars. Furthermore, grammar engineering mostly concerns the evolution of grammars and migration of words, but largely ignores the adaptation of processing tools.

Refactoring: High level programming languages offer facilities such as methods and classes for abstraction creation. Since the invariance dilemma also holds for these abstraction mechanisms, classes and methods too have to evolve over time to adapt to changes to the variability / invariance separation. Refactoring is the restructuring of programs by "altering its internal structure without changing its external behavior"[4]. This limitation of expressiveness allows for a high level of automation. Several state of the art development environments such as *Eclipse*[10] implement a number of refactorings that automate the adaptation of abstraction instances to changes to abstraction specifications.
Some refactorings, i.e. `rename class/method`, `introduce/remove parameter` [4] can be used to change the parameters offered by classes and methods and thus for evolution of variability in abstraction specifications.
The concept of refactoring has been applied to numerous languages [13] and even to meta models [18]. While refactorings are a help for mastering variability evolution, they are certainly not sufficient due to their limited expressiveness.

Feature Model Synchronization: In [7], Kim and Czarnecki propose an approach to adapt feature configurations to changes to their feature model that can arise during product line evolution. While they state promising results for feature model synchronization, they do not address the problem of tool adaptation.

Language Evolution: In [6] we present the *Lever*[11], a tool for the evolutionary development of (domain specific) languages. Lever provides evolution operations on the language specification level, that automate the migration of existing words and the adaptation of the compiler / generator for the language. Lever is currently implemented as a research prototype that will be released beginning of 2007. We believe that many of the concepts that Lever employs for the evolution of languages are also applicable for the evolution of other abstraction mechanisms.
TransformGen [5] is an earlier approach to language evolution that was developed in the context of syntax directed editors. It cannot deal with arbitrary syntax and ignores the adaptation of language processing tools, though.

To the best of our knowledge, these approaches consider the mechanics of evolution of their abstraction mechanisms in relative isolation. We are not aware of an underlying theory of abstraction or variability evolution. However, we are convinced that this problem is so fundamental to software reuse and variability management that it deserves consideration on a general level.

5. Conclusion

The amount of variable and invariant information in an abstraction specification — independent of the actual abstraction mechanism used — determines the reuse benefit that the creation of an abstraction brings to software development. Unfortunately, variability is a two-edged sword: too much variability results in too low a level of abstraction. Too little variability on the other hand reduces the number of times an abstraction can be reused and thus also ruins the reuse benefit of an abstraction.

The optimal partition between variable and invariant information can only be determined with respect to all cases in which an abstraction is to be applied. Since this is impossible in practice — because future use cases are unknown — abstractions can only achieve optimal reuse benefit, if they consider the currently known use cases and are adapted when new use cases arise.

Since the evolution of an abstraction affects its existing instances and processing tools, evolution of abstractions is only feasible if the compensational effort is automated to a high degree. Approaches to automate this compensational effort have been developed in relative isolation for numerous abstraction mechanisms. These approaches tend to reinvent the wheel, since the underlying problem of abstraction evolution is common among all abstraction mechanisms. We are convinced, that a unified analysis — across different abstraction mechanisms — is required to tackle the problem at its core. We hope that results obtained this way will be beneficial for many abstraction mechanisms, including the ones mentioned in this paper.

6 Acknowledgements

The authors would like to thank Birgit Penzenstadler, Florian Deißenböck, Stefan Wagner and Martin Feilkas for

[10] www.eclipse.org
[11] Stands for "Language Evolver"

inspiring discussions on the topic and helpful comments on the paper.

References

[1] Scott W. Ambler and Pramodkumar J. Sadalage. *Refactoring Databases: Evolutionary Database Design (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2006.

[2] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 311–322, New York, NY, USA, 1987. ACM Press.

[3] Kajal T. Claypool, Jing Jin, and Elke A. Rundensteiner. Serf: schema evolution through an extensible, re-usable and flexible framework. In *CIKM '98: Proceedings of the seventh international conference on Information and knowledge management*, pages 314–321, New York, NY, USA, 1998. ACM Press.

[4] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[5] David Garlan, Charles W. Krueger, and Barbara Staudt Lerner. Transformgen: automating the maintenance of structure-oriented environments. *ACM Trans. Program. Lang. Syst.*, 16(3):727–774, 1994.

[6] Elmar Juergens and Markus Pizka. The Language Evolver Lever - Tool Demonstration. In John Boyland and Anthony Sloane, editors, *Proceedings of the Sixth Workshop on Language Descriptions, Tools and Applications*, pages 62–67, 2006.

[7] Chang Hwan Peter Kim and Krzysztof Czarnecki. Synchronizing cardinality-based feature models and their specializations. In *ECMDA-FA*, pages 331–348, 2005.

[8] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, 2005.

[9] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[10] Ralf Lämmel. Grammar Adaptation. In *Proc. Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*, pages 550–570. Springer-Verlag, 2001.

[11] Ralf Lämmel. Coupled Software Transformations (Extended Abstract). In *First International Workshop on Software Evolution Transformations*, November 2004.

[12] Ralf Lämmel and Wolfgang Lohmann. Format Evolution. In *Proc. 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, volume 155 of *books@ocg.at*, pages 113–134. OCG, 2001.

[13] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.

[14] Peter Naur and Brian Randell, editors. *Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968*. Scientific Affairs Division, NATO, 1969.

[15] Markus Pizka and Elmar Juergens. Tool Supported Multi Level Language Evolution. *To Appear*, 2007.

[16] Stefan Wagner and Florian Deissenboeck. Language development is software design. *To Appear*, 2006.

[17] P Wegner. Varieties of reusability. In *Workshop on Reusability in Programming*, 1983.

[18] Jing Zhang, Yuehua Lin, and Jeff Gray. Generic and domain-specific model refactoring using a model transformation engine.