Matti Rossi
Jeff Gray
Jonathan Sprinkle
Juha-Pekka Tolvanen (eds.)

Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM´09)

HELSINGIN KAUPPAKORKEAKOULU
HELSINKI SCHOOL OF ECONOMICS

B-108

Matti Rossi, Jonathan Sprinkle, Jeff Gray, Juha-Pekka Tolvanen (eds.)

Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)

HELSINGIN KAUPPAKORKEAKOULUN

JULKAISUJA

B-108

ISSN 0356-889X
ISBN 978-952-488-371-9

E-versio:
ISBN 978-952-488-372-6

Helsingin kauppakorkeakoulu -
HSE Print 2009

Welcome to the 9th Workshop on Domain-Specific Modeling Workshop – DSM'09

Preface

Domain-Specific Modeling (DSM) is continuing to receive interest among the general software engineering community. As an example, the special issue of *IEEE Software* (July/August 2009) gave the approach much needed visibility. Several controlled experiments have shown that DSMs are more productive than general model based approaches. As Booch et al. have stated, "the full value of MDA is only achieved when the modeling concepts map directly to domain concepts rather than computer technology concepts." For example, DSM for cell phone software would have concepts like "Soft key button," "SMS" and "Ring tone," and generators to create calls to the corresponding code components.

Continued investigation is still needed in order to advance the acceptance and viability of DSM. This workshop, which is in its ninth incarnation at OOPSLA 2009, features research and experience papers describing new ideas at either a practical or theoretical level. On the practical side, several papers in these proceedings describe application of modeling techniques within a specific domain. As in previous workshops, there are plenty of language examples contributed to these proceedings.

We have organized the 18 papers in these proceedings to emphasize general areas of interest into which the papers loosely fit. Authors from both industry and academia have contributed research ideas that initiate and forward the technical underpinnings of domain-specific modeling. The papers in this proceedings are categorized into the areas of *Language Design, Language Examples, DSLs for the web, Transformations and Language Evolution, Model Verification and Testing* and *Special Topics*. Many papers in these proceedings are cross-cutting in their analysis and reporting. As a whole, the body of work highlights the importance of metamodeling and related tooling, which significantly ease the implementation of domain-specific languages and provide support for experimenting with the modeling language as it is built (thus, metamodel-based language definition also assists in the task of constructing generators that reduce the burden of tool creation and maintenance).

We hope that you will enjoy this record of the workshop and find the information within these proceedings valuable toward your understanding of the current state-of-the-art in Domain-Specific Modeling.

Matti Rossi, Jonathan Sprinkle, Jeff Gray, Juha-Pekka Tolvanen

October 2009
Orlando, Florida

9th WORKSHOP ON DOMAIN-SPECIFIC MODELING

25-26 October 2009, Orlando, USA

Table of Contents

Design Guidelines for Domain Specific Languages

Gabor Karsai
Institute for Software Integrated Systems
Vanderbilt University
Nashville, USA

Holger Krahn
Software Engineering Group
Department of Computer Science
RWTH Aachen, Germany

Claas Pinkernell
Software Engineering Group
Department of Computer Science
RWTH Aachen, Germany

Bernhard Rumpe
Software Engineering Group
Department of Computer Science
RWTH Aachen, Germany

Martin Schindler
Software Engineering Group
Department of Computer Science
RWTH Aachen, Germany

Steven Völkel
Software Engineering Group
Department of Computer Science
RWTH Aachen, Germany

ABSTRACT

Designing a new domain specific language is as any other complex task sometimes error-prone and usually time consuming, especially if the language shall be of high-quality and comfortably usable. Existing tool support focuses on the simplification of technical aspects but lacks support for an enforcement of principles for a good language design. In this paper we investigate guidelines that are useful for designing domain specific languages, largely based on our experience in developing languages as well as relying on existing guidelines on general purpose (GPLs) and modeling languages. We defined guidelines to support a DSL developer to achieve better quality of the language design and a better acceptance among its users.

1. INTRODUCTION

Designing a new language that allows us to model new technical properties in a simpler and easier way, describe or implement solutions, or to describe the problem resp. requirements in a more concise way is one of the core challenges of computer science. The creation of a new language is a time consuming task, needs experience and is thus usually carried out by specialized language engineers. Nowadays, the need for new languages for various growing domains is strongly increasing. Fortunately, also more sophisticated tools exist that allow software engineers to define a new language with a reasonable effort. As a result, an increasing number of DSLs (Domain Specific Languages) are designed to enhance the productivity of developers within specific domains. However, these languages often fit only to a rather specific domain problem and are neither of the quality that they can be used by many people nor flexible enough to be easily adapted for related domains.

During the last years, we developed the frameworks MontiCore [13] and GME [2] which support the definition of domain specific languages. Using these frameworks we designed several DSLs for a variety of domains, e.g., a textual version of UML/P notations [17] and a language based on function nets in the automotive domain [5]. We experienced that the design of a new DSL is a difficult task because different people have a varying perception of what a "good" language actually is.

This of course also depends on the taste of the developer respectively the users, but there are a number of generally acceptable guidelines that assist in language development, making it more a systematic, methodological task and less an intellectual ad-hoc challenge. In this paper we summarize, categorize, and amend existing guidelines as well as add our new ones assuming that they improve design and usability of future DSLs.

In the following we present general guidelines to be considered for both textual and graphical DSLs with main focus is on the former. The guidelines are discussed sometimes using examples from well-known programming languages or mathematics, because these languages are known best. Depending on the concrete language and the domain these guidelines have to be weighted differently as there might be different purposes, complexity, and number of users of the resulting language. For example, for a rather simple configuration language used in only one project a timely realization is usually more important than the optimization of its usability. Therefore, guidelines must be sometimes ignored, altered, or enforced. Especially quality-assurance guidelines can result in an increased amount of work.

While we generally focus in our work on DSLs that are specifically dedicated to modeling aspects of (software) systems, we believe that these guidelines generally hold for any DSL that embeds a certain degree of complexity.

1.1 Literature on Language Design

For programming languages, design guidelines have been intensively discussed since the early 70s. Hoare [8] introduced simplicity, security, fast translation, efficient object code, and readability as general criteria for the design of good languages. Furthermore, Wirth [22] discussed several guidelines for the design of languages and corresponding compilers. The rationale behind most of the guidelines and hints of both articles can be accepted as still valid today, but the technical constraints have changed dramatically since the 70s. First of all, computer power has increased significantly. Therefore, speed and space problems have become less important. Furthermore, due to sophisticated tools (e.g., parser generators) the implementation of accompanying tools is often not a necessary part of the language development any more. Of course, both articles concentrate on programming languages and do not consider the greater variety of domain specific languages.

More recently, authors have also discussed the design of domain specific modeling languages. General principles for modeling language design were introduced in [14]. These include simplicity, uniqueness, consistency, and scalability, on which we will rely later. However, the authors did not discuss how these higher level principles can be achieved. In [12] certain aspects of the DSL development are explained and some guidelines are introduced. More practical guidelines for implementing DSLs are given in [10]. These focus on how to identify the necessary language constructs to generate full code from models. The authors explain how to provide tool support with the MetaEdit+ environment. [20] explains 12 lessons learned from DSL experiments that can help to improve a DSL. Although more detailed discussions on explicit guidelines are missing, these lessons embed documented empirical evidence – a documentation that many other discussions, including ours do not have. In [16] the authors introduce a toolset which supports the definition of DSLs by checking their consistency with respect to several objectives. Language designers can select properties of their DSL to be developed and the system automatically derives other design decisions in order to gain a consistent language definition. However, the introduced criteria cover only a subset of the decisions to be made and hence, cannot serve as the only criteria for good language design. Quite the contrary, to our experience many design guidelines cannot be translated in automatic measures and thus cannot be checked by a tool.

1.2 Categories of DSL Design Guidelines

The various design guidelines we will discuss below, can be organized into several categories. Essentially, these guidelines describe techniques that are useful at different activities of the language development process, which range from the domain analysis to questions of how to realize the DSL to the development of an abstract and a concrete syntax including the definition of context conditions. An alignment of guidelines with the language development activities and the developed artifacts has the advantage that a language designer can concentrate on the respective subset of the guidelines at each activity. This should help identifying and realizing the desired guidelines. Therefore, we decided for a development phase oriented classification and identified the following categories:

Language Purpose discusses design guidelines for the early activities of the language development process.

Language Realization introduces guidelines which discuss how to implement the language.

Language Content contains guidelines which focus on the elements of a language.

Concrete Syntax concentrates on design guidelines for the readable (external) representation of a language.

Abstract Syntax concentrates on design guidelines for the internal representation of a language.

For each of these categories we will discuss the design guidelines we found useful. Please be aware that the subsequently discussed guidelines sometimes are in conflict with each other and the language developer sometimes has to balance them accordingly. Additionally, semantics is explicitly not listed as a separate step as it should be part of the entire development process and therefore has an influence on all of the categories above.

2. DSL DESIGN GUIDELINES

2.1 Language Purpose

Language design is not only influenced by the question of what it needs to describe, but equally important what to do with the language. Therefore, one of the first activities in language design is to analyze the aim of the language.

Guideline 1: *"Identify language uses early."* The language defined will be used for at least one task. Most common uses are: documentation of knowledge (only) and code generation. However, there are a lot more forms of usage: definition or generation of tests, formal verification, automatic analysis of various kinds, configuration of the system at deployment- or run-time, and last but increasingly important, simulation.

An early identification of the language uses have strong influence on the concepts the language will allow to offer. Code generation for example is not generally feasible when the language embeds concepts of underspecification (e.g., nondeterministic Statecharts). Even if everything is designed to be executable, there are big differences regarding the overhead necessary to run certain kinds of models. If efficient execution on a small target machine is necessary (e.g., mobile or car control device) then high-level concepts must be designed for optimized code generations. For simulation and validation of requirements however, efficiency plays a minor role.

Guideline 2: *"Ask questions."* Once the uses of a language have been identified it is helpful to embed these forms of language uses into the overall software development process. People/roles have to be identified that develop, review, and deploy the involved programs and models. The following questions are helpful for determining the necessary decisions: Who is going to model in the DSL? Who is going to review the models? When? Who is using the models for which purpose?

Based thereon, the question after whether the language is too complex or captures all the necessary domain elements can be revisited. In particular, appropriate tutorials for the DSL users in their respective development process should now be prepared.

Guideline 3: *"Make your language consistent."* DSLs are typically designed for a specific purpose. Therefore, each feature of a language should contribute to this purpose, otherwise it should be omitted. As an illustrative example we consider a platform independent modeling language. In this language, all features should be platform independent as well. This design principle was already discussed in [14].

2.2 Language Realization

When starting to define a new language, there are several options on how to realize it. One can implement the DSL from scratch or reuse and extend or reduce an existing language, one can use a graphical or a textual representation, and so on. We have identified general hints which have to be taken into account for these decisions.

Guideline 4: *"Decide carefully whether to use graphical or textual realization."* Nowadays, it is common to use tools supporting the design of graphical DSLs such as the Eclipse Modeling Framework (EMF) or MetaEdit+. On the other hand, there exist sophisticated tools and frameworks like MontiCore or xText for text-based modeling. As described in [6], there are a number of advantages and disadvantages for both approaches. Textual representations for example usually have the advantage of faster development and are platform and tool independent whereas graphical models provide a better overview and ease the understanding of models. Therefore, advantages and disadvantages have to be weighted and matched against end users' preferences in order to make a substantiated decision for one of the realizations. From this point on, a more informed decision can be made for a concrete tool to realize the language based on their particular features and the intended use of the language. Comparisons can be found in [21] or [3].

Guideline 5: *"Compose existing languages where possible."* The development of a new language and an accompanying toolset is a labor-intensive task. However, it is often the case that existing languages can be reused, sometimes even without adaptation. A good example for language reuse is OCL: it can be embedded in other languages in order to define constraints on elements expressed in the hosting language.

The most general and useful form of language reuse is thus the unchanged embedding of an existing language into another language. A more sophisticated approach is to have predefined holes in a host language, such that the definition of a new language basically consists of a composition of different languages. For textual languages this compositional style of language definitions is well understood and supported by sophisticated tools such as [11] which also assists the composition of appropriate tools.

However, according to the seamlessness principle [14], the concepts of the languages to be composed need to fit together. In the UML, the object oriented paradigm underlies both class diagrams and Statecharts which therefore fit well together. Additionally, when composing languages care must be exercised to avoid confusion: similar constructs with different semantics should be avoided.

Guideline 6: *"Reuse existing language definitions."* If the language cannot be simply composed from some given language parts, e.g., by language embedding as proposed in guideline 5, it is still a good idea to reuse existing language definitions as much as possible. In [18] more possible realization strategies, such as language extension or language specialization are analyzed. This means, taking the definition of a language as a starter to develop a new one is better than creating a language from scratch. Both the concrete and the abstract syntax will benefit from this form of reuse. The new language then might retain a look-and-feel of the original, thus allowing the user to easily identify familiar notations. Looking at the abstract syntax of existing languages, one can identify "language pattern" (quite similar to design pattern), which are good guidelines for language design. For example, expressions, primary expressions, or statements have quite a common pattern in all languages.

Only if there is no existing language/notation or the disadvantages do not allow using the strategies mentioned above, a standalone realization should be considered. The websites of parser generators like Antlr [1] or Atlantic Zoo [19] are a good starting point for reusing language definitions.

Guideline 7: *"Reuse existing type systems."* A DSL used for software development often comprises and even extends either a property language such as OCL or an implementation language such as Java. As described in [8], the design of a type system for such a language is one of the hardest tasks because of the complex correlations of name spaces, generic types, type conversions, and polymorphism.

Furthermore, an unconventional type system would be hard for users to adopt as well. Therefore, a language designer should reuse existing type systems to improve comprehensibility and to avoid errors that are caused by misinterpretations in an implementation. Furthermore, it is far more economical to use an existing type system, than developing a new one as this is a labor intensive and error-prone task. A well-documented object-oriented type system can be tailored to the needs of the DSL or even an implemented reusable type system can be used (e.g. [4]).

2.3 Language Content

One main activity in language development is the task of defining the different elements of the language. Obviously, we cannot define in general which elements should be part of a language as this typically depends on the intended use. However, the decisions can be guided by some basic hints we propose in this Section.

Guideline 8: *"Reflect only the necessary domain concepts."* Any language shall capture a certain set of domain artifacts. These domain artifacts and their essential properties need to be reflected appropriately in the language in a way that the language user is able to express all necessary domain concepts. To ensure this, it is helpful to define a few models early to show how such a reflection would look like. These models are a good basis for feedback from domain experts which helps the developer to validate the language definition against the domain. However, when designing a language not all domain concepts need to be reflected, but only those that contribute to the tasks the language shall be used for.

Guideline 9: *"Keep it simple."* Simplicity is a well known criterion which enhances the understandability of a language [8, 14, 22]. The demand for simplicity has several reasons. First, introducing a new language in a domain produces work in developing new tools and adapting existing processes. If the language itself is complex, it is usually harder to understand and thus raises the barrier of introducing the language. Second, even when such a language is successfully introduced in a domain, unnecessary complexity still minimizes the benefit the language should have yielded. Therefore, simplicity is one of the main targets in designing languages. The following more detailed Guidelines 10, 11, and 12 will show how to achieve simplicity.

Guideline 10: *"Avoid unnecessary generality."* Usually, a domain has a finite collection of concepts that should be reflected in the language design. Statements like "maybe we can generalize or parameterize this concept for future changes in the domain" should be avoided as they unnecessarily complicate the language and hinder a quick and successful introduction of the DSL in the domain. Therefore, this guideline can also be defined as "design only what is necessary".

Guideline 11: *"Limit the number of language elements."* A language which has several hundreds of elements is obviously hard to understand. One approach to limit the number of elements in a language for complex domains is to design sublanguages which cover different aspects of the systems. This concept is, e.g., employed by the UML: different kinds of diagrams are used for special purposes such as structure, behavior, or deployment. Each of them has its own notation with a limited number of concepts.

A further possibility to limit the number of elements of a language is to use libraries that contain more elaborated concepts based on the concepts of the basic language and that can be reused in other models. Elements which were previously defined as part of the language itself can then be moved to a model in the library (compare, e.g., I/O in Pascal vs. C++). Furthermore, users can extend a library by their own definitions and thus, can add more and more functionality without changing the language structure itself. Therefore, introducing a library leads to a flexible, extensible, and extensive language that nevertheless is kept simple. On the other hand, a language capable of library import and definition of those elements must have a number of appropriate concepts embedded to enable this (e.g., method and class definitions, modularity, interfaces - whatever this means in the DSL under construction). This principle has successfully been applied in GPL design where the languages are usually small compared to their huge standard libraries.

Guideline 12: *"Avoid conceptual redundancy."* Redundancy is a constant source of problems. Having several concepts at hand to describe the same fact allows users to model it differently. The case of conceptual richness in C++ shows that coding guidelines then usually forbid a number of concepts. E.g., the concept of classes and structs is nearly identical, the main difference is the default access of members which is `public` for structs and `private` for classes. Therefore, classes and structs can be used interchangeably within C++ whereas the slight difference might be easily forgotten. So, it should be generally avoided to add redundant concepts to a language.

Guideline 13: *"Avoid inefficient language elements."* One main target of domain specific modeling is to raise the level of abstraction. Therefore, the main artifacts users deal with are the input models and not the generated code. On the other hand, the generated code is necessary to run the final system and more important, the generated code determines significant properties of the system such as efficiency. Hence, the language developer should try to generate efficient code.

Furthermore, efficiency of a model should be transparent to the language user and therefore should only depend on the model itself and not on specific elements used within the model. Elements which would lead to inefficient code should be avoided already during language design so that only the language user is able to introduce inefficiency [8]. For example, in Java there is no operator to get all instances of one class as this would increase memory usage and operating time significantly. However, this functionality can be implemented by a Java user if needed.

2.4 Concrete Syntax

Concrete syntax has to be chosen well in order to have an understandable, well structured language. Thus, we concentrate on the concrete syntax first and will deal with the abstract syntax later.

Guideline 14: *"Adopt existing notations domain experts use."* As [20] says, it is generally useful to adopt whatever formal notation the domain experts already have, rather than inventing a new one.

Computer experts and especially language designers are usually very practiced in learning new languages. On the contrary, domain experts often use a language for a longer time and do not want to learn a new concrete syntax especially when they already have a notation for a certain problem. As already mentioned, it is often the case that the introduction of a DSL makes new tools and modified processes necessary. Inventing a new concrete syntax for given concepts would raise the barrier for domain experts. Thus, existing notations should be adopted as much as possible. E.g., queries within the database domain should be defined with SQL instead of inventing a new query language. Even if queries are only part of a new language to be defined SQL could be embedded within the new language.

In case a suitable notation does not already exist, the new language should be adopted as close as possible to other existing notations within the domain or to other common used languages. A good example for commonly accepted languages are mathematical notations like arithmetical expressions [8].

Guideline 15: *"Use descriptive notations."* A descriptive notation supports both learnability and comprehensibility of a language especially when reusing frequently-used terms and symbols of domain or general knowledge. To avoid misinterpretation it is highly important to maintain the semantics of these reused elements. For instance, the sign "+" usually stands for addition or at least something semantically similar to that whereas commas or semicolons are interpreted as separators. This applies to keywords with a widely-accepted meaning as well. Furthermore, keywords should be easily identifiable. It is helpful to restrict the number of keywords to a few memorizable ones and of course, to have a keyword-sensitive editor.

A good example for a descriptive notation is the way how special character like Greek letters are expressed in Latex. Instead of using a Unicode-notation each letter can be expressed by its name (`\alpha` for α, `\beta` for β, and so on).

Guideline 16: *"Make elements distinguishable."* Easily distinguishable representations of language elements are a basic requirement to support understandability. In graphical DSLs, different model elements should have representations that exhibit enough syntactic differences to be easily distinguishable. Different colors as the only criteria may be counterproductive, e.g., when printed in black and white. In textual languages usually keywords are used in order to separate kinds of elements. These keywords have to be placed in appropriate positions of the concrete syntax, as otherwise readers need to start backtracking when "parsing" the text [8, 22]. The absence of keywords is often based on efficiency for the writer. But this is a very weak reason because models are much more often read than written and therefore to be designed from a readers point of view.

Guideline 17: *"Use syntactic sugar appropriately."* Languages typically offer syntactic sugar, i.e., elements which do not contribute to the expressiveness of the language. Syntactic sugar mainly serves to improve readability, but to some extent also helps the parser to parse effectively. Keywords chosen wisely help to make text readable. Generally, if an efficient parser cannot be implemented, the language is probably also hard to understand for human readers.

However, an overuse of the addition of syntactic sugar distracts, because verbosity hinders to see the important content directly. Furthermore, it should be kept in mind that several forms of syntactic sugar for one concept may hinder communication as different persons might prefer different elements for expressing the same idea.

Nevertheless the introduction of syntactic sugar can also improve a language, e.g., the enhanced for-statement in Java 5 is widely accepted although it is conceptually redundant to a common for-statement. This is a conflict to guideline 12, but the frequency of occurrence of common for-statements in Java legitimates a more effective alternative of this notation.

Guideline 18: *"Permit comments."* Comments on model elements are essential for explaining design decisions made for other developers. This makes models more understandable and simplifies or even enables collaborative work. So a widely accepted standard form of grouped comments, like `/* ... */`, and line comments, like `// ...` for textual languages or text boxes and tooltips for graphical languages should be embedded.

Furthermore, specially structured comments can be used for further documentation purposes as generating HTML-pages like Javadoc. In [8] it is mentioned that the "purpose of a programming language is to assist in the documentation of programs". Therefore we recommend that every DSL should allow a user to generally comment at various parts of the model. If desired, the language may even contain the definition of a comment structure directly, thus enforcing a certain style of documentation.

Guideline 19: *"Provide organizational structures for models."* Especially for complex systems the separation of models in separate artifacts (files) is inevitable but often not enough as the number of files would lead to an overflowed model directory. Therefore, it is desirable to allow users to arrange their models in hierarchies, e.g., using a package mechanism similar to Java and store them in various directories.

As a consequence, the language should provide concepts to define references between different files. Most commonly "`import`" is used to refer to another name space. Imports make elements defined in other DSL artifacts visible, while direct references to elements in other files usually are expressed by qualified names like "`package.File.name`". Sometimes one form of import isn't enough and various relations apply which have to be reflected in the concrete syntax of the language.

Guideline 20: *"Balance compactness and comprehensibility."* As stated above, usually a document is written only once but read many times. Therefore, the comprehensibility of a notation is very important, without too much verbosity. On the other hand, the compactness of a language is still a worthwhile and important target in order to achieve effectiveness and productivity while writing in the language. Hence a short notation is more preferable for frequently used elements rather than for rarely used elements.

Guideline 21: *"Use the same style everywhere."* DSLs are typically developed for a clearly defined task or viewpoint. Therefore, it is often necessary to use several languages to specify all aspects of a system. In order to increase understandability the same look-and-feel should be used for all sublanguages and especially for the elements within a language. In this way the user can obtain some kind of intuition for a new language due to his knowledge of other ones. For instance, it is hardly intuitive if curly braces are used for combining elements in one language and parentheses in another. Additionally, a general style can also assist the user in identifying language elements, e.g., if every keyword consists of one word and is written in lower case letters.

A conflicting example is the embedment of OCL. One the one hand it is possible to adapt the OCL syntax to the enclosing language to provide the same syntactic style in both languages. On the other hand different OCL styles impede the comprehensibility of OCL, what endorses the use of a standard OCL syntax.

Guideline 22: *"Identify usage conventions."* Preferably not every single aspect should be defined within the language definition itself to keep it simple and comprehensible (see guideline 11). Furthermore, besides syntactic correctness it is too rigid to enforce a certain layout directly by the tools. Instead, usage conventions can be used which describe more detailed regulations that can, but need not be enforced.

In general, usage conventions can be used to raise the level of comprehensibility and maintainability of a language. The decision, whether something goes as a usage convention or within a language definition is not always clear. So, usage conventions must be defined in parallel to the concrete syntax of the language itself. Typical usage conventions include notation of identifiers (uppercase/lowercase), order of elements (e.g. attributes before methods), or extent and form of comments. A good example for code conventions for a programming language can be found in [9].

2.5 Abstract Syntax

Guideline 23: *"Align abstract and concrete syntax."* Given the concrete syntax, the abstract syntax and especially its structure should follow closely to the concrete syntax to ease automated processing, internal transformations and also presentation (pretty printing) of the model.

In order to align abstract and concrete syntax three main principles apply: First, elements that differ in the concrete syntax need to have different abstract notations. Second, elements that have a similar meaning can be internally represented by reusing concepts of the abstract syntax (usually through subclassing). This is more a semantics-based decision than a structurally based decision. Third, the abstract notation should not depend on the context an element is used in but only on the element itself. A pretty bad example for context-dependent notations is the use of "`=`" as assignment in OCL-statements (let-construct) and as equality in OCL-expressions. Here, the semantics obviously differs whilst the syntax is equal.

Furthermore, the use of a transformation engine usually also requires an understanding of the internal structure of a language, which is related to the abstract syntax. Therefore, the user to some extent is exposed to the internal structure of the language and hence needs an alignment between his concrete representations and the abstract syntax, where the transformations operate on.

Alignment of both versions of syntax and the seamlessness principle discussed in [14] assures that it is possible to map abstractions from a problem space to concrete realizations in the solution space. For a domain specific language the domain is then reflected as directly as possible without much bias, e.g., of implementation or executability considerations.

Guideline 24: *"Prefer layout which does not affect translation from concrete to abstract syntax."* A good layout of a model can be used to simplify the understanding for a human reader and is often used to structure the model. Nevertheless, a layout should be preferred which does not have any impact on the meaning of the model, and thus, does not affect the translation of the concrete to the abstract syntax and the semantics. As an example, this is the case for computer languages where the program structure is achieved by indentation. From a practical point of view, line separators, tabs, and spaces are often treated differently depending on editors and platforms and are usually difficult to distinguish by a human reader. If these elements gain a meaning, developers have to be much more cautious and a collaborative development requires more effort. For graphical languages a well-known bad example is the twelve o'clock semantics in Stateflow [7] where the order of the placement of transitions can change the behavior of the Statechart. To simplify the usage of DSLs, we recommend that the layout of programs doesn't affect their semantics.

Guideline 25: *"Enable modularity."* Nowadays, systems are very complex and thus, hard to understand in their entirety. One main technique to tackle complexity is modularization [15] which leads to a managerial, flexible, comprehensible, and understandable infrastructure. Furthermore, modularization is a prerequisite for incremental code generation which in turn can lead to a significant improvement of productivity. Therefore, the language should provide a means to decompose systems into small pieces that can be separately defined by the language users, e.g., by providing language elements which can be used in order to reference artifacts in other files.

Guideline 26: *"Introduce interfaces."* Interfaces in programming languages provide means for a modular development of parts of the system. This is especially important for complex systems as developers may define interfaces between their parts to be able to exchange one implementation of an interface with another which significantly increases flexibility. Furthermore, the introduction of interfaces is a common technique for information hiding: developers are able to change parts of their models and can be sure that these changes do not affect other parts of the system when the interface does not change. Therefore, we recommend that a DSL should provide an interface concept similar to the interfaces of known programming languages.

One example of interfaces are visibility modifiers in Java. They provide a means to restrict the access to members in a simple way. Another common example are ports, e.g., in composite structure diagrams, which explicitly define interaction points and specify services they provide or need, thus declaring a more detailed interface of a part of a system.

3. DISCUSSION

In the previous sections we introduced and categorized a bundle of guidelines dedicated to different language artifacts and development phases. Some of them already contained notes on relationships with other guidelines and trade-offs between them, and some of them briefly discussed their importance in different project settings. However, the following more detailed discussion shall help to identify possible conflicting guidelines and their reasons and gives hints on decision criteria.

The most contradicting point is reuse of existing artifacts versus the implementation of a language from scratch (cf. No. 5, 6, and 7). The main reason for the reuse of a language or a type system is that it can significantly decrease development time. Furthermore, existing languages often provide at least an initial level of quality. Thus, some of the guidelines, e.g., guidelines which target at consistency (e.g., No. 21) or claim modularity (e.g., No. 25), are met automatically. However, reusing existing languages can hinder flexibility and agility as an adaption may be hard to realize if not impossible. The same ideas apply to an improvement of the reused language itself (e.g., to meet guidelines which were not respected by the original language): the implementation of a single guideline may require a significant change of the language. Another important point is that this approach may influence the satisfiability of other guidelines. One example is No. 14 which suggests the reuse of existing notations of the domain. In case there are no languages which are similar to these notations, this guideline and language reuse are obviously contradicting. Furthermore, combining several existing languages may introduce conceptual inconsistencies, such as different styles or different underlying type systems which have to be translated into each other (cf., No. 5).

Implementing a new language from scratch in turn permits a high degree of freedom, agility, and flexibility. In this case, some guidelines can be realized more easily than in the case of reuse. However, these advantages are not for free: designing concrete and abstract syntax, context conditions, and a type system are time- and cost-intensive task. To summarize, a decision whether to reuse existing languages or to implement a new one is one of the most important and critical decisions to be made.

Another important point which was already mentioned in the introduction is that some of the presented guidelines have to be weighted according to the project settings, to the form of use, etc. One example is the expected size of the languages instances. Some DSLs serve as configuration languages and thus, typical instances consist of a small amount of lines only. Other DSLs are used to describe complex systems leading to huge instances. In the former case guidelines which target at compositionality or claim references between files (e.g., No. 19 and 25) have nearly no validity whereas in the latter example these guidelines are of high importance. However, not only the expected size of the instances can influence the weight of guidelines. Another important aspect is the intended usage of the language. Sometimes DSLs are not executable; they are designed for documentation only. In these cases, the guideline which demands to avoid inefficient elements in the language (No. 13) is of course not meaningful. However, for languages which are translated into running code, this is of high importance.

A last point we want to discuss here are the costs induced by applying the guidelines. Some of them can be implemented easily and straightforward (e.g., distinguishability of elements or permitting comments, No. 16 and 18) whilst others require a significant amount of work (e.g., introduction of references between files including appropriate resolution mechanisms and symbol tables, No. 19). Of course, especially guidelines whose implementation is cost intensive have to be matched against project settings as described above. For small DSLs such guidelines should be ignored instead as the cost will often not amortize the improvements. However, from our experiences DSLs are often subject to changes. While growing these guidelines become more and more important. The main problem which emerges in these cases is that adding new things to a grown language (e.g., modularity) is typically more difficult and time-consuming than it would have been at the beginning. Therefore, analyzing the domain and usage scenarios as described in Guidelines 1 and 2 can prevent those unnecessary costs.

4. CONCLUSION

In this paper 26 guidelines have been discussed that should be considered while developing domain specific languages. To our experience this set of guidelines is a good basis for developing a language. For space reasons, we restricted ourselves to guidelines for designing the language itself. Other guidelines are needed for successfully integrating DSLs in a software development process, deploying it to new users, and evolving the syntax and existing models in a coherent way.

In general, a guideline should not be followed closely, but many of them are proposals as to what a language designer should consider during development. Some of the guidelines have to be discussed in certain domains, because they might not have the same relevance and as discussed many guidelines contradict each other and the language developer has to balance them appropriately.

But generally, the consideration of explicitly formulated guidelines is improving language design. We also think that it is worthwhile to develop much more detailed sets of concrete instructions for particular DSLs. We currently focus on textual languages in the spirit of Java.

Although we have compiled this list from literature and our own experience, we are sure that this list is not complete and has to be extended constantly. In addition, guidelines might change during time as developers gather more experience, tools become more elaborate, and taste changes. Maybe some guidelines are not relevant anymore in a few years, as some guidelines from the 1970's are less important today.

Acknowledgment: The work presented in this paper is partly undertaken in the MODELPLEX project. MODELPLEX is a project co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2002-2006). Information included in this document reflects only the authors' views. The European Community is not liable for any use that may be made of the information contained herein.

5. REFERENCES

[1] Antlr Website `www.antlr.org`.

[2] GME Website `http://www.isis.vanderbilt.edu/projects/gme/`.

[3] T. Goldschmidt, S. Becker, and A. Uhl. Classification of concrete textual syntax mapping approaches. In *ECMDA-FA*, pages 169–184, 2008.

[4] J. Gough. *Compiling for the .NET Common Language Runtime (CLR).* Prentice Hall, November 2001.

[5] H. Grönniger, J. Hartmann, H. Krahn, S. Kriebel, and B. Rumpe. View-based modeling of function nets. In *Proceedings of the Object-oriented Modelling of Embedded Real-Time Systems (OMER4) Workshop, Paderborn,*, October 2007.

[6] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering*, 2007.

[7] G. Hamon and J. Rushby. An operational semantics for stateflow. In *Fundamental Approaches to Software Engineering: 7th International Conference (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 229–243, Barcelona, Spain, March 2004. Springer-Verlag.

[8] C. A. R. Hoare. Hints on programming language design. Technical report, Stanford University, Stanford, CA, USA, 1973.

[9] Java Code Conventions `http://java.sun.com/docs/codeconv/`.

[10] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley, 2008.

[11] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In *Proceedings of Tools Europe*, 2008.

[12] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. Technical Report SEN-E0309, Centrum voor Wiskunde en Informatica, Amsterdam, 2005.

[13] MontiCore Website `http://www.monticore.de`.

[14] R. Paige, J. Ostroff, and P. Brooke. Principles for Modeling Language Design. Technical Report CS-1999-08, York University, December 1999.

[15] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[16] P. Pfahler and U. Kastens. Language Design and Implementation by Selection. In *Proc. 1st ACM-SIGPLAN Workshop on Domain-Specific-Languages, DSL '97*, pages 97–108, Paris, France, January 1997. Technical Report, University of Illinois at Urbana-Champaign.

[17] B. Rumpe. *Modellierung mit UML.* Springer, Berlin, May 2004.

[18] D. Spinellis. Notable Design Patterns for Domain Specific Languages. *Journal of Systems and Software*, 56(1):91–99, Feb. 2001.

[19] The Atlantic Zoo Website `http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/`.

[20] D. Wile. Lessons learned from real DSL experiments. *Science of Computer Programming*, 51(3):265–290, June 2004.

[21] D. S. Wile. Supporting the DSL Spectrum. *Computing and Information Technology*, 4:263–287, 2001.

[22] N. Wirth. On the Design of Programming Languages. In *IFIP Congress*, pages 386–393, 1974.

Evaluating the Use of Domain-Specific Modeling in Practice

Juha Kärnä
Polar Electro
Professorintie 5
FI-90440 Kempele, Finland
+358 8 5202 100
Juha.Karna@polar.fi

Juha-Pekka Tolvanen
MetaCase
Ylistönmäentie 31
FI-40500 Jyväskylä, Finland
+358 14 641 000
jpt@metacase.com

Steven Kelly
MetaCase
Ylistönmäentie 31
FI-40500 Jyväskylä, Finland
+358 14 641 000
stevek@metacase.com

ABSTRACT

Domain-Specific Modeling (DSM) raises the level of abstraction beyond coding, making development faster and easier. When companies develop their own in-house DSM solution — domain-specific modeling languages and code generators — they often need to provide evidence that it gives better results than their current practice. We describe an approach applied at Polar to evaluate a DSM solution for developing embedded devices. The evaluation approach takes into account the objectives set for the creation of the DSM solution and collects data via controlled laboratory studies. The evaluation proved the benefits of the DSM solution: an increase of at least 750% in developer productivity, and greatly improved quality of the code and development process.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**] Design Tools and Techniques - *user interfaces, state diagrams* D.2.6 [**Software Engineering**] Programming Environments - *programmer workbench, graphical environments* D.3.2 [**Programming Languages**] Language Classifications - *Specialized application languages, very high-level languages*

General Terms

Design, Economics, Experimentation, Languages.

Keywords

Domain-specific modeling, code generation, empirical evaluation, language design

1. INTRODUCTION

Domain-Specific Modeling (DSM) improves on current software development approaches in two ways. First, it raises the level of abstraction beyond programming by specifying the solution in languages that directly uses concepts and rules from a specific problem domain. Second, it can generate fully functional production code from these high-level specifications. The most effective DSM solutions are usually applied within a single company. The domain can then be narrowed and the automation becomes easier to achieve when addressing the requirements of only one company.

When a company moves from coding to DSM the fundamental questions are: will the DSM solution provide the desired benefits, and can those benefits be measured? Development teams in companies, however, do not usually have the time and resources to conduct extensive analysis, such as building the same system twice with different development approaches, using parallel teams [2], evaluating dozens of developers [1], analyzing large numbers of development tasks [2], or focusing on development activities in detail with video recording, speaking while working, or observing individual developers' actions [6]. Many good scientific research methods are simply too expensive and time-consuming for practical use in a commercial setting. Some of the characteristics of good empirical research, like a large number of participants to support generalization of the results, are not always even possible since there may only be a handful of developers using the particular language within the company.

The evaluation of the DSM solution may not even be necessary at all if a small inspection already shows a major difference: "why conduct a comparison when we can see that a task that earlier took days can be done with DSM during an afternoon?" The comparison is not always so straightforward. The development team may need to present more compelling data to management to get resources for finalizing the DSM solution or investing in training and tools. The nature of the work may be such that there is no clear view on the current development process, e.g. it is scattered among teams. The last situation is typical if the DSM solution reduces duplication and unnecessary work by changing the roles and division of work among teams or even organizations.

This paper presents the evaluation of a DSM solution at Polar. The evaluation approach combines developers' opinions with quantative measurements of the development process. We first introduce the domain for which our case's DSM solution was created: UI applications in sports heart rate monitors [4]. We briefly describe the DSM solution and show a sample model to illustrate the modeling language. Then we move to the actual evaluation and describe the evaluation criteria and how the evaluation was conducted. We report the findings: at least a 750% increase in productivity, with developers also estimating the quality of the code and the quality of the design process to be significantly better with DSM. We conclude by proposing some improvements for evaluating DSM in companies: gathering metrics stepwise starting from initial prototypes, and considering development processes outside the typical implementation phase.

2. DOMAIN

The study was conducted at Polar, the leading brand in the sports instruments and heart rate monitoring category, delivering state-of-the-art training technology and solutions. This study focused on heart rate monitors. Figure 1 illustrates three typical products in this product category. The features in these products depend on the product segment and the type of sports the product is designed for, such as running, cycling, fitness and cross-training, team sports or snow sports. Some possible features in these products include:

- Heart rate measurement, analysis and visualization
- Calorie calculation, e.g. current, cumulative, expenditure rate, active time
- Speed: current, average, maximum
- Distance, based on interval, trip, recovery
- Altimeter, vertical speed, altitude alarms, slope counter, graphical trend
- Cycling information, e.g. pedaling rate and cycling power
- Barometer, pressure drop alarm, graphical trend
- Compass
- Temperature
- Odometer
- Logbooks
- Exercise diaries
- Sensor connectivity (heart rate, speed, cadence, power, GPS)
- Data transfer for web and other applications
- Date and weekday indicator
- Localization with different display texts
- Visual and audible alarm in target zones

Depending on the features there are also various settings, starting from age and weight to bicycle wheel size adjustment and various exercise settings and plans. These products also show time with various time related applications, such as dual time zone, stopwatch, alarm, countdown timer and lap time.

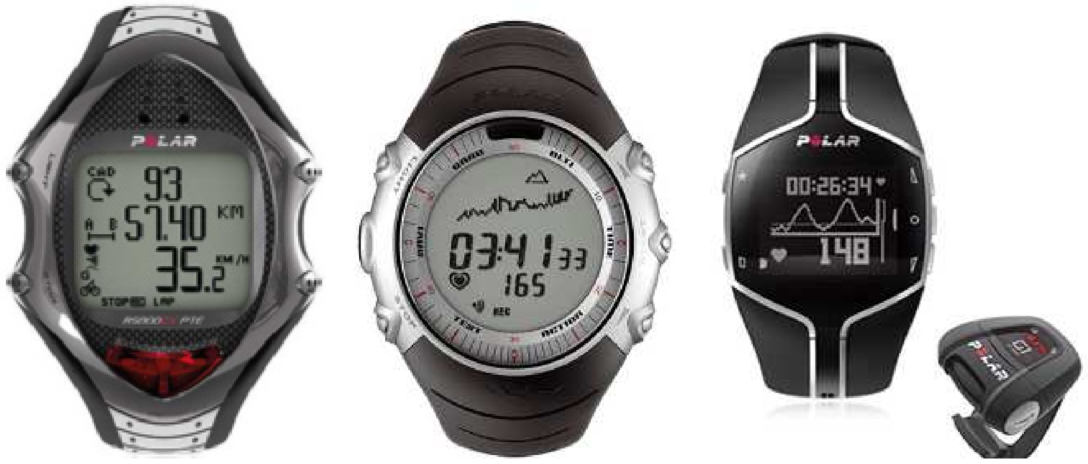


Figure 1. Sample products

Software development for these devices is constrained by the limited resources they contain, such as the amount of memory, processor speed and battery life. The actual area of interest — the domain — reported in this study is the UI applications: how the various capabilities and features are available to the user. The sample products in Figure 1 give some indication of what UI applications can look like as they show the display and its content in different applications. UI applications, however, do not focus on (G)UI elements alone. They also cover control, navigation, and connectivity to other devices, such as to sensors and other applications to transfer the data. The design and implementation of the UI applications is heavily constrained by device capabilities such as display size, type, and user interaction controls. It is worth mentioning that as these devices are used in special conditions — users may have little time and concentration capability while exercising — the usability of UI applications is crucial.

3. THE DSM SOLUTION

When implementing the DSM solution Polar decided to focus on UI applications for two main reasons. First, the UI applications form the single largest piece of software, typically requiring 40–50% of the development time. Improvements to UI application development would therefore have the greatest impact on overall development times. Second, the analysis of the domain showed that 70% of UI applications would be easy to automate with DSM, while a further 25% could probably also be handled with DSM. This left only 5% of the UIs that would be difficult to cover with DSM, indicating that the domain was understood well enough to specify the languages and code generators.

Polar set a number of requirements for the DSM solution. These included:

1. Fundamentally improve the productivity of UI application development
2. Significantly reduce the manual work needed to copy data from specifications into code
3. Be independent of the target environment
4. Be independent of the programming language, but support currently used languages such as C and Assembler
5. Make the introduction of new developers easier
6. Be usable for both experienced and novice developers

7. Improve the quality and maintainability of the code
8. Be easy to modify to meet new and changing requirements, e.g. when resources in the device change

At Polar, one UI application developer defined the modeling language, along with the generators that transformed models made with that language into the artifacts the company needed (e.g. code, configuration files, links to simulators, document generation). The modeling language was supported by a tool [5] that provided the functionality needed to work effectively with models, such as reusing models, refactoring and replacing model elements, organizing and handling large models, multi-user access — as well as usual modeling operations like copy and paste.

UI application developers can thus use this modeling language and tool to create high-level models, such as Figure 2. This model shows a small sample feature for selecting a favorite drink: a selection state along with two views ('Water', 'Milk') as well as various navigation paths within the application. The diagram uses a small portion of the modeling language: the full set of modeling concepts are shown in the toolbar. These concepts originate from the problem domain and thus the modeling language raises the abstraction from coding, while also providing support for reuse when developing multiple products. The diagram is also executable, in that full code can be automatically generated from it.

While the application in Figure 2 illustrates the use of the language, it is about the smallest possible model. In real cases there may be dozens of elements in a diagram, dozens of diagrams in an application, and dozens of applications in a full product. An element in one diagram can be linked, referred to and reused in other diagrams, or can be linked to a subdiagram specifying it in more detail. Applications too can be reused between products.

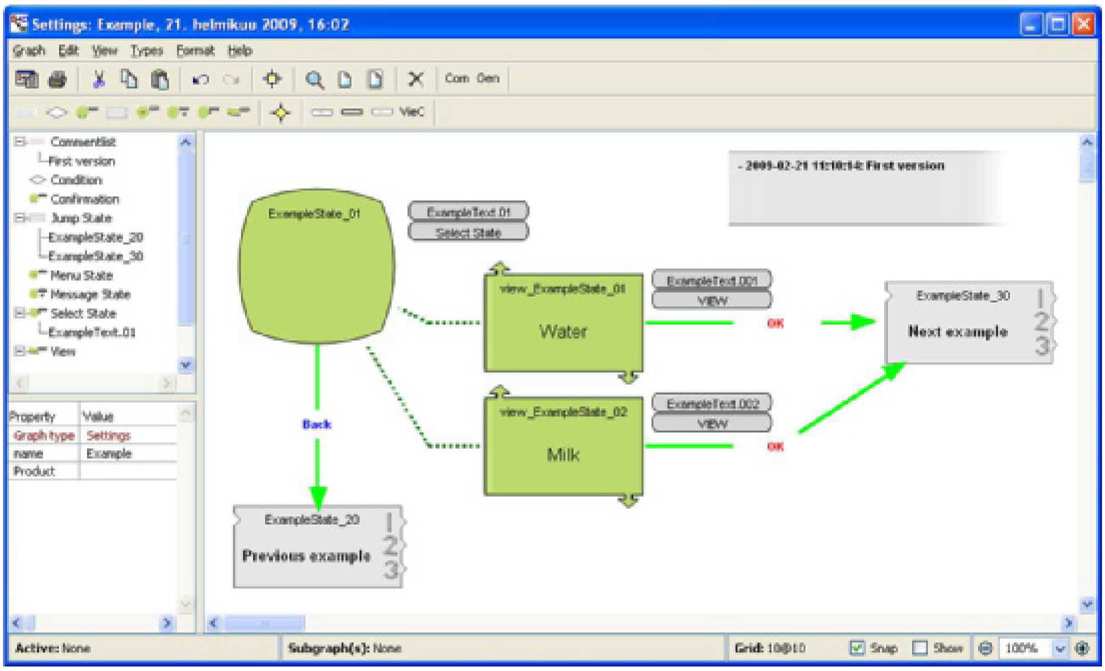


Figure 2. Sample model of a UI application.

While the whole lifecycle of product development was acknowledged and known, the DSM solution focused on technical design and implementation. In other words, the primary users of the language and generators described in the paper are the current UI application developers. This means that the expected outcome of the generators was the full code of the UI applications, which earlier had to be written by hand. Other artifacts than code can also be generated from the same models, e.g. documentation, build scripts and material for review meetings, saving the UI developers further time.

In addition to serving UI application implementation, generators could also be created to support other roles and processes in the life cycle: Generators can provide input for testing, parts of the user manuals, or rapid prototyping as part of user interface and interaction design, typically carried out before the implementation phase. Limited space does not allow us to go into these details and the evaluation reported in this paper addresses only the technical design and implementation tasks.

4. EVALUATING THE DSM SOLUTION

With their evaluation Polar wanted to find out how well, if at all, the requirements set for the created DSM solution were met. The evaluation was made by using the DSM solution in product development, covering the application design and implementation phases. Development tasks were carried out using the modeling language to create models and the generator to produce the application code. The starting point for DSM use during the evaluation was a UI specification, as used in the current development process. The evaluation therefore did not test the possible scenario of using the DSM solution further upstream at the UI specification phase. Similarly links to other development phases, like testing, localization, documentation and providing user manuals, were excluded from the evaluation: although DSM could help there too, the current DSM solution offers at least the same output to those phases as earlier manual coding.

Before the evaluation, the creator of the DSM solution had already used it to build example models during its creation. During a pilot project he had also implemented the majority of a whole product's UI applications, including some large ones.

The evaluation focused on three factors: developer productivity, product quality and the general usability of the tooling. These factors also formed the major requirements for the DSM solution as outlined in Section 3. The measures for these factors were selected so that they could be easily understood and estimated by the developers. To calculate the return on investment — when the effort to define the language and generators is amortized — the application development time was recorded in addition to asking developers opinions on the possible influence to productivity. The evaluation did not evaluate if the requirements of independency of target environment (#3) and of generated programming language (#4) were met as the generators were made only for one target and programming language applied in the company. As the support of customizable code generators for different targets and programming languages is well attested, these requirements were not further analyzed.

The evaluation was set up to find credible and repeatable results with reasonable costs. Rather than developing a whole product, Polar set up a laboratory experiment to develop one typical UI application: the setup for sporting exercises. Experience from the pilot project allowed the size and complexity of this application to be chosen such that it was expected to be completed with the DSM solution within a few hours. Results of the single UI application development were then compared to the development approach currently in use, and to the experiences of modeling on a larger scale in the pilot project.

In the laboratory experiment the same UI application was developed separately by 6 developers. The developers were selected so that they all had experience of making UI applications. They could then compare the DSM approach with the current development approach. Four of the developers had over three years' experience in UI application development; the other two had less than one year's experience. Only one of the developers had previous experience with the modeling tool used.

4.1 Evaluation process

The evaluation process had four phases: training, conducting the laboratory experiment, evaluating the correctness of the results and reporting experiences. Training covered introduction to the modeling language and to the modeling tool. Since the language concepts were taken directly from the problem domain, and hence already familiar to the developers, training took 1 hour. In this time the basic modeling features of the tool were also taught.

The input for the development task in the laboratory experiment was the specification of the desired exercise setup UI application. The developers were each timed separately as they modeled the application. They were asked to finish the task as completely as possible, and the completeness and correctness of the result were checked together with the developer. If there were errors or data was missing the specification or the modeling language was explained so that the developer could finish the implementation.

Finally, the developers' experiences and opinions were collected with a questionnaire and with interviews. The results are described in the following sections.

4.2 Development time and productivity

The influence on productivity (requirement #1) was inspected in two ways: by measuring the development time and by collecting developers' opinions after having used both approaches: the current development method and the DSM approach used for the first time.

Development time for the UI application varied among the developers from 75 minutes to 125 minutes, with a mean of 105 minutes. Implementing the same UI application with the current development approach would take about 960 minutes (16 hours). The productivity improvement for the mean time is thus over 900%. Even for the slowest completion time, the productivity increase is over 750%.

The pilot project had produced UI applications whose implementation time with the current development approach was estimated to have taken 3 weeks (120 hours). The size of the UI application models in the experiment was measured to be 16% of the total size of the pilot project, based on the number of states and views in the models. This gives us a second way to estimate the time to code this UI application, 16% of 120 hours = 1152 minutes. Taking the mean of the two estimates, 1056 minutes, gives a mean productivity increase over the 6 developers of over 1000%.

The influence on productivity was also measured by asking the developers' opinions — after all, they now had experience of using both approaches. As shown in Figure 3, there were almost no differences among developers' opinions: all found the DSM approach to be significantly faster than current practice. Developers' opinions were asked on a scale from 1 to 5, with 5 being the best. Although the laboratory experiment did not cover maintenance (new features and error corrections), developers were also asked if the DSM solution would support maintenance better than the current approach: 5 developers thought DSM would be better and one could not say.

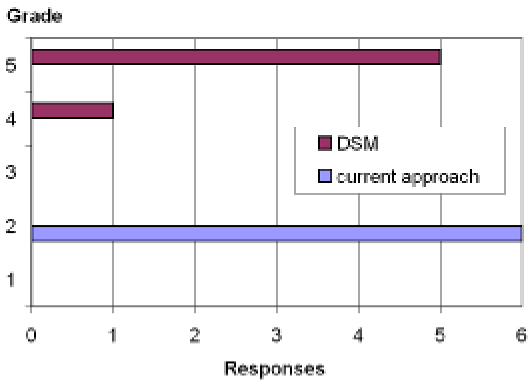


Figure 3. Perceived productivity (scale 1–5, 5=best productivity).

4.3 Quality of process and resulting code

When studying the influence on quality, both process and result were evaluated (requirement #7). The influence on the process was evaluated by asking developers' opinions on how well the development approaches — current and DSM — prevented errors. As with the results of the productivity measurement there was a clear difference in DSM's favor, although the answers varied more (Figure 4). The piloting of the DSM solution also showed that the DSM solution's support for error prevention could be further improved. For example, the DSM solution did not check that values entered as text met a specific syntax (using regular expressions in MetaEdit+ [5]), and some fields used string entries when selection lists would better ensure correctness. Also, model checking did not inspect all relevant parts of model completeness and errors. These areas for improvement will be taken into account in future versions of the DSM solution, and the error prevention grades are expected to improve as a result.

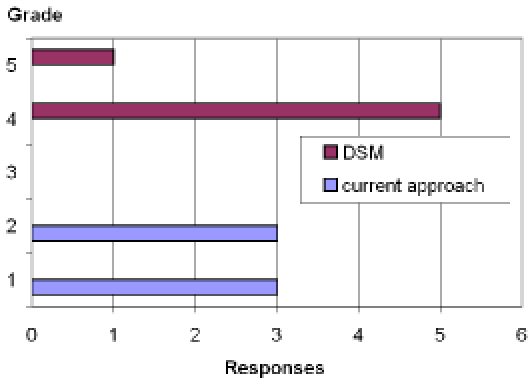


Figure 4. Error prevention

The quality of the outcome was measured by inspecting the generated code and comparing it with the manually written code. Code quality is particularly relevant for embedded products like heart rate monitors. The results show that the generated code was considered to be of better quality: a smaller, but still clear, difference between the approaches (Figure 5).

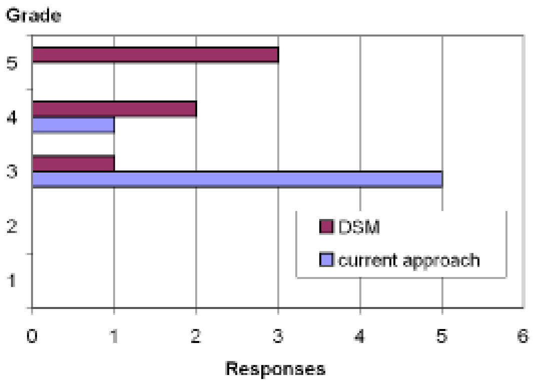


Figure 5. Code quality.

4.4 Usability and learning

To assess the usability (requirement #6) developers were asked how usable they found the resulting modeling tools and how easy it was to learn and use the modeling language. The answers were then compared to the evaluation of the current approach. Figure 6 shows the results on usability. Here the opinions of developers differed the most, but the created DSM tooling (average 4.5) was still considered clearly better than current tools (average 2.5).

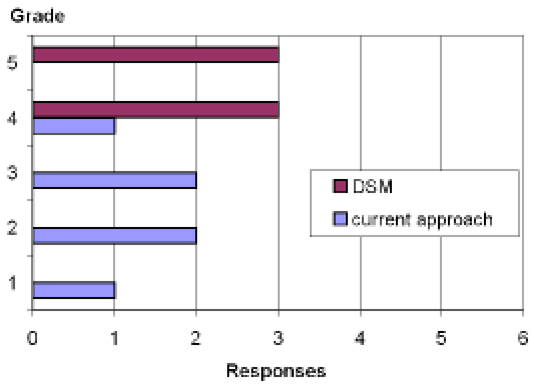


Figure 6. Tool usability.

Since none of the developers was a beginner the study did not directly measure how well new developers could learn the DSM approach (requirement #5). Introducing new developers just for the sake of DSM evaluation was not considered practical. Instead, developers estimated the ease of learning. The results indicated that learning the UI application design and implementation with DSM would be much easier than with the current approach. As Figure 7 indicates this opinion was quite clear.

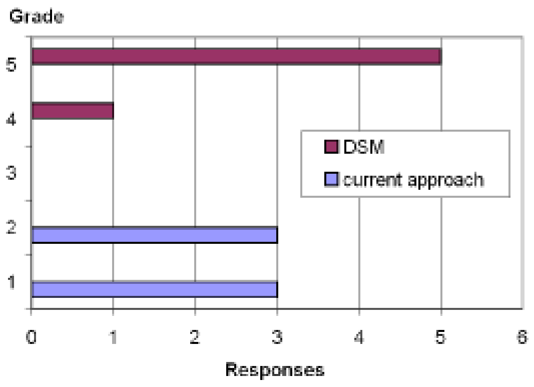


Figure 7. Ease of learning.

5. RETURN ON INVESTMENT

The benefits of DSM do not come for free: first the modeling language and generators, the DSM solution, must be developed. Depending on the tooling used, time may also need to be allocated to tool creation and maintenance.

At Polar, creation of the DSM solution took 7.5 working days, covering the development of the modeling language and the code generator. Both of these were implemented using MetaEdit+ Workbench [5]. MetaEdit+ automatically provides modeling tools based on the modeling language, so no extra time needed to be spent on tool building. It is worth noting that the 7.5 days also included the creation of example models specifying UI applications, along with related code. This was natural since the best way to test a DSM solution under development is to apply it immediately on real examples.

When we compare the time to implement the DSM solution to the productivity improvements when creating UI applications, it is evident that the investment would pay back very quickly, as illustrated in Figure 8. The pilot project was estimated to be about 64% of a whole product, so a whole product would take over 23 days to build with the current development method. With DSM, after the 7.5 days' metamodeling, the first whole product would take 2.3 days to build, making DSM over twice as fast as coding even for the first product. Each subsequent product would take another 2.3 days, so in the time it took to build one whole product by coding, Polar could build several whole products with DSM.

The time required to build the UI applications for a complete product may seem to become almost trivial. However in reality, the problem domain is not completely static. Therefore after the pilot project it is essential to evolve the DSM solution further to maintain the measured benefits. From our experiences in other languages [3], after the first few products the effort to maintain the DSM solution becomes a small fraction of the time to develop each product.

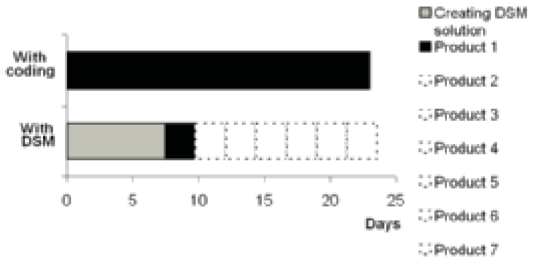


Figure 8. Return on investment: comparison.

6. CONCLUDING REMARKS

We described an approach and results to evaluate a particular DSM solution. The evaluation showed that the DSM solution for developing UI applications for heart rate monitors is applicable for its domain. The applicability was inspected with a pilot project, laboratory experiment and questionnaire. In the pilot project the majority of a whole product was developed with the DSM solution. In the laboratory experiment, the DSM solution was found to be at least 7.5 times and on average 10 times as productive as the current development approach. In the questionnaire, the DSM solution was considered to offer better productivity, quality and usability, and be easier to learn. Figure 9 summarizes the questionnaire findings by comparing the current approach and DSM based on the average grading calculated from developers' opinions.

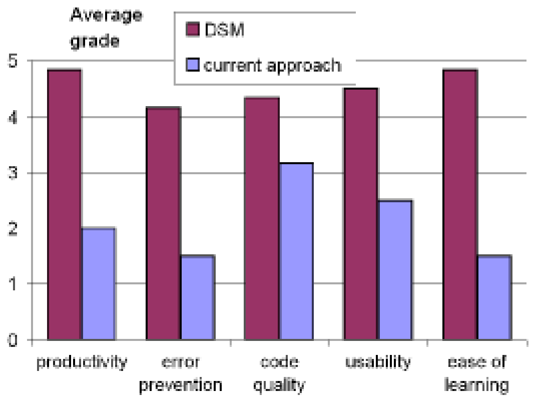


Figure 9. Comparing approaches based on average grades.

While the actual evaluation focused on the laboratory experiment and questionnaire, the DSM solution was also evaluated during its construction and in the pilot project, which developed a large portion of a whole product. The collection of data could already have been started with those initial prototypes, so that development time statistics could be measured from a wider variety of modeling tasks. A further point of evaluation would be to extend the scope of the DSM solution to cover a larger part of the development processes, from requirements and UI specification steps to build automation and testing. This would allow the same domain concepts to be applied pervasively within the company through the modeling languages. Parts of these steps could also be automated with generators, saving time and avoiding manual errors when copying data from one step to another (requirement #2). The DSM solution evaluated here is thus not final and complete, but can be extended incrementally in the future. One obvious way is to extend the language to include future new UI concepts. This need for extensibility was actually one requirement (#8) that was not evaluated here, because of the focus on a single product and its set of UI concepts. One way to evaluate the extensibility would be to apply the DSM solution to model older generation products and study if their development could be supported.

Since companies have limited resources to evaluate new approaches in practice, the evaluation approach described strikes a balance between the effort expended on the evaluation and the credibility of the results achieved. It was considered particularly important to have several developers involved in the evaluation, as this improved the visibility of the DSM solution within the company and the credibility of its evaluation. It also helped to train the developers and offered the possibility to obtain feedback for further improvements. While the results are not statistically significant or generalizable, they are highly relevant and credible for the company performing the evaluation. The evaluation approach itself can be used to evaluate other kinds of DSM solutions and in other companies. In that case, the main foreseeable changes would be adaptations to the questionnaire to ensure it covers the issues most relevant to that company's development.

7. REFERENCES

[1] Cao, L., Ramesh, B., Rossi, M., Are Domain Specific Models Easier to Maintain Than UML Models?, IEEE Software, July/August, 2009

[2] Kieburtz, R. et al., A Software Engineering Experiment in Software Component Generation, Proceedings of 18th International Conference on Software Engineering, Berlin, IEEE Computer Society Press, 1996

[3] Kelly, S., Tolvanen, J-P., Domain-Specific Modeling: Enabling Full Code Generation, Wiley-IEEE Society Press, 2008

[4] Kärnä, J., Using DSM for embedded UI development (in Finnish), Master's thesis, University of Oulu, 2009

[5] MetaCase, MetaEdit+ Workbench 4.5 SR1 User's Guide, http://www.metacase.com/support/45/manuals/, 2008

[6] Wijers, G., Modeling Support in Information Systems Development, Thesis Publishers Amsterdam, 1991

Multi-Language Development of Embedded Systems

Thomas Kuhn
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern
+49 631 6800 2177
thomas.kuhn@iese.fraunhofer.de

Soeren Kemmann
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern
+49 631 6800 2218
soeren.kemmann@iese.fraunhofer.de

Mario Trapp
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern
+49 631 6800 2272
mario.trapp@iese.fraunhofer.de

Christian Schäfer
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern
+49 631 6800 2121
christian.schaefer@iese.fraunhofer.de

ABSTRACT

Graphical, well focused and intuitive domain specific languages (DSLs) are more and more used to design parts of embedded systems. These languages are highly specialized and often tailored to one domain; one single language therefore cannot describe all relevant aspects of systems and system components. This raises the need for heterogeneous modeling approaches that are capable of combining multiple DSLs into holistic system models. Our CompoSE modeling approach focuses on this problem; it does not only cover system modeling with DSLs, but provides also interfacing of language specific generators and harmonization of generated code. In this paper, we describe the principles of CompoSE, together with the integration of an existing modeling language with industrial strength tool support into CompoSE. Supporting the integration of existing languages is of particular importance in the domain of embedded systems, because modern modeling approaches will only be accepted in industry if they support existing and proven technologies.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General – *System specification methodology*

General Terms

Design, Languages

Keywords

System modeling, Domain specific languages, Multi formalism development

1. INTRODUCTION

Development of embedded systems in research and industry is more and more shifting from code based development to model driven development (MDD) approaches, which are founded on high-level modeling languages. Modeling languages are not as generic as general purpose programming languages, they provide more specialized language constructs, e.g. for the creation of data flow based systems (cf. Simulink) or for the creation of system models (cf. SysML). These MDD approaches are supported by industrial strength tool chains; prominent examples of MDD tools that are applied in both research and industry are Simulink, ASCET, SCADE, Rhapsody, Artisan, and MagicDraw. MDD tools implement modeling languages, provide infrastructure support, e.g. tailored editors and code generators, and include runtime libraries and frameworks that support execution of generated code. Domain specific languages (DSLs) are more specialized than generic MDD approaches; being tailored to a specific application domain, they enable domain experts to express themselves with native constructs of their respective domains. One example for DSLs is a graphical language for creating wiring diagrams. These modeling languages are either implemented as language profiles, e.g. as UML profiles, or they are built on top of existing language frameworks (cf. Eclipse GMF or MetaEdit+). In both cases, DSLs provide their own tool chains and model to code transformations.

This is a major challenge for the development of embedded systems: generic and domain specific modeling languages are limited and support some aspects of embedded system development only. Simulink for example supports definition of data flow based behavior only, UML based languages support the definition of software architecture and control flow, and SysML supports the definition of system architectures. Graphical editors, code generators, and language frameworks only support one or a limited set of modeling languages. Detailed modeling of all aspects of complex embedded systems therefore requires the combination of models defined in multiple modeling languages and tool chains to provide one holistic system model. Code generation needs to be done with multiple independent generators in this case. This yields the situation that developers need to combine multiple generated artifacts and runtime libraries, and need to connect required inputs and provided outputs of models, which may even implement different semantics. One common execution model is required that supports all relevant modeling languages. This non-trivial task currently limits the applicability of DSL approaches in development of complex software systems, since the effort required for integrating modeling languages may outweigh the additional benefits of modeling languages.

CompoSE is our multi formalism modeling approach that supports the integration of modeling languages on language, infrastructure and runtime levels. Being independent of concrete modeling languages, it defines a common host component model, which supports multi-formalism development of embedded systems. Guest modeling languages are integrated as language components; these languages are applied for modeling functional and non-functional details of system components. Support is provided for the integration of general purpose, and domain specific modeling languages. Language components address all of the three aforementioned layers: they provide integration of modeling languages at language, infrastructure, and runtime levels.

The remainder of this paper explains CompoSE principles and is structured as following: Section 2 describes the basic principles of our multi-formalism development approach. Section 3 describes the CompoSE host language in greater detail. Section 4 provides an application example that illustrates briefly the integration of Simulink into CompoSE. Section 5 surveys and discusses related approaches. Section 6 draws conclusions and lays out future work.

2. MULTI-FORMALISM DEVELOPMENT

CompoSE supports multi-formalism development through the integration of independent modeling languages and tools into one multi formalism framework (*MFF*). Despite the independency of the different languages and tools, the MFF ensures their seamless combination for the creation of integrated system models. This is achieved through the application of component based development basic principles to the domain of language engineering.

CompoSE is based on the principle of one host language and several guest languages. The host language defines system components and a basic set of views for modeling system architectures; it also defines language constructs for the integration of language components. Language components integrate guest languages into CompoSE. As shown in Figure 1, language components address the three main elements of modeling languages to ensure their seamless integration: view types provide integration on language level, infrastructure interfaces integrate tool chains, and runtime interfaces ensure interfacing of generated code with a common runtime model.

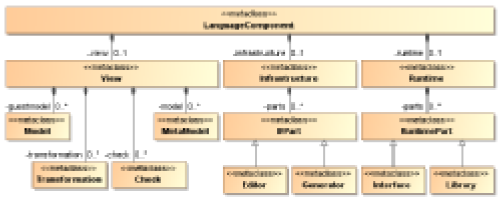


Figure 1: Language component meta model

Language components provide one or multiple view types that integrate modeling languages. Compose additionally provides several predefined view types that support definition of interfaces (*InterfaceView*), aggregations (*AggregationView*), and coupling (*CouplingView*). Native models of integrated existing modeling languages are stored in guest models, which are containers that conform to some unknown, guest specific format, and are therefore not directly accessible. Meta models support the (bidirectional) projection of guest model parts into the host model through transformations – this way, information stored in guest models is made accessible, and is shared and synchronized between guest models and language components. Additionally, views represent their modeling languages to developers and therefore support the manipulation of their underlying models – for this reason, infrastructure parts (*IFParts*) are used to expose language infrastructure, e.g. graphical editors.

Infrastructure parts (*IFParts*) enable the integration of existing language tool chains. These parts implement proxies that provide common interfaces to the CompoSE MFF and hide native interfaces of language specific tools. Runtime parts of language components define the runtime interfaces of generated code; when existing tool chains are integrated into CompoSE, they model the interface between generated code and existing, language specific runtime frameworks. A CompoSE runtime framework then provides glue code that interfaces generated code for each language component with each other and that conforms to a common runtime specification. Note that CompoSE does not include a specific runtime environment, but it defines common requirements that conforming runtime environments need to fulfill. These requirements define syntactic and semantic constraints that runtime framework implementations need to conform to. Adapter code that is generated by generator proxies serves as interface between the generated code from language specific tools (whose interface is defined through language components) and the runtime framework. Figure 2 provides an example – two host components are defined: the component *Control* that realizes a data low based controlling algorithm, and a *Filter* component that preprocesses data for the *Control* component. The *Control* component is realized with Simulink, the *Filter* component is realized with a domain specific language. Therefore, two language components provide necessary views, infrastructure, and runtime support.

The Simulink language component provides the Simulink realization view, integrates the native Simulink tool chain, which consists of a code generator (*Simulink Generator*) and of the Simulink runtime framework. It also includes the *Simulink Proxy* that generated adapter code (*Simulink Adapter*), which interfaces generated code by Simulink with the common runtime framework. The DSL language component provides a view that supports editing models based on its domain specific language together with a code generator. No proxies and adapters are required, since the code generator outputs conforming code directly.

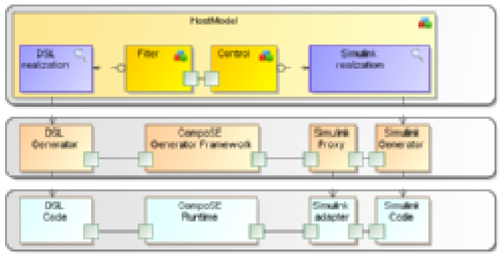


Figure 2: Compose Multi Formalism Framework

3. THE COMPOSE HOST LANGUAGE

The CompoSE host language implements a component modeling approach that is based on components, properties, ports and links. Components represent parts of the developed system, which are either black or white boxes. Ports belong to components and define points of interaction that links are connected to. Properties store component information – three types of properties are defined by CompoSE: Guest model properties store complete guest models in their native format. Meta model properties are based on a CompoSE conforming meta model definition, and represent containers that store models conforming to those meta models. Primitive properties store one type, e.g. an integer or a structured type. Properties are subdivided into two property types: Specification properties define whole or partial component specifications. Component specifications define what a component does, and how a component is to be used. Specification properties are associated with specification views. Realization properties define how a component is realized – they are associated to realization views. Component realizations always need to conform to the specification of their component.

3.1 CompoSE language components

Language components integrate new modeling or domain specific languages into CompoSE that are used for defining component details. On language level, language components consist of views, transformations, and models.

Views present data, which is represented by models. For this reason, two model types are distinguished: guest models and meta models. Existing modeling languages that ship with their own tool chains usually store data in their own container format, e.g. a language specific binary representation. Models stored in such a container are referred to as guest models. Other language components cannot access this data, since file format and structure is not known – guest model properties are therefore black boxes for other language components. New, CompoSE conforming DSLs store all of their data in containers that conform to defined meta models instead, therefore, this data may be accessed by other language components – this is a white box representation.

3.1.1 *Guest model synchronization*

If a language component uses a guest model representation for storing models, these models are not accessible for other language components, which may be cumbersome. For example, a Simulink view defines component realizations as data flow between input and output flow ports of components. The *InterfaceView* (see below), which is a predefined and therefore language independent view of CompoSE, defines component ports as part of the component interface as well. Both views therefore store the same information in different properties: the Simulink view stores component ports as part of its Simulink guest model, the interface view stores component ports in a meta model property.

This situation is not satisfactory for developers using CompoSE – they need to manually ensure consistency between views. Existing tools for UML for example provide this synchronization between diagrams that operate on the same model automatically – changes in the model through one diagram are immediately reflected in all other diagrams. CompoSE provides a similar functionality through transformations in a manner that supports multiple modeling languages. Transformation components implement model to model transformations; they are part of language component views and therefore implement a bridge between host and guest models. Transformations may be applied to transform models conforming to one meta model into a model that conforms to another meta model of the same component, to modify models, and to transform models into guest models and back. Guest models may only be accessed by the language component that defines them, and each guest model type may be defined by one language component only. Through transformations, complete guest models or parts of it are projected into models that conform to defined meta models, and are therefore accessible by other language components (see Figure 3).

In the example defined by Figure 3, two views are attached to the system component type *Control*. This component type has three properties – the first property *Interface.interface* defines the component interface and contains data conforming to the interface meta model defined by the common *InterfaceView*. It is manipulated through the *interface specification* view. The *Simulink.simulink* property holds the guest model of the Simulink realization, and is manipulated through the Simulink realization view. Model transformations synchronize the Simulink guest model and the interface meta model with the Simulink flow meta model, which is a common white box representation. This model is not manipulated directly through a view, and stored in the *Simulink.flow* property.

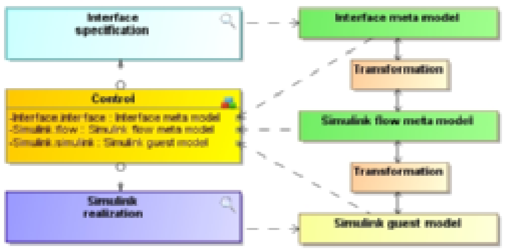


Figure 3: Components, models, views, and properties

As shown in the example, component data is stored in properties. Therefore, complex components possibly require a large number of properties to store the data of all views. Additional data that is shared between views, for example ports, attributes, and operations are stored in properties as well. Therefore, to prevent namespace pollution, the name of a property is composed out of a language component that its type belongs to, together with its identifier (see Figure 3).

3.1.2 *Specialization*

Language component hierarchies support the concept of specialization, which is known from other languages, e.g. from the MOF or from the UML. However, specialization of language components needs different semantics to ensure proper handling of views, infrastructure, and runtime. Specialized language components inherit all elements of more generic components and may override them. Specialized components, for example, define new guest models, new transformations, and new meta models. Existing transformations and meta models are possibly extended by specialized language components. Infrastructure parts of parents may be inherited or overridden. In the latter case, the existing infrastructure (tools, editors, code generators…) of the parent may be used by the infrastructure of the specialized language component. Runtime interfaces may be inherited or overridden, but overriding is only permitted with more specialized interfaces that at least provide the functionality of the base interface type. Figure 6 illustrates an example for language component specialization. The base component *GenericLanguageComponent* defines a framework for all subsequent language component definitions. The component *DataflowLanguage* redefines the language view and the runtime. The *DataflowView* view adds a data flow meta model, the *DataflowRuntime* component adds a data flow runtime interface. The Simulink language component extends all three views. Therefore, all existing elements are inherited first. The meta model *SimulinkMM* may only extend the more generic *DataflowMM* meta model, since it is its specialization. The guest model definition is a new language component element. Simulink infrastructure are new language component elements as well. The Simulink runtime interface *SimulinkRuntimeIF* replaces the old *DataflowRuntimeIF* with a derived and specialized interface.

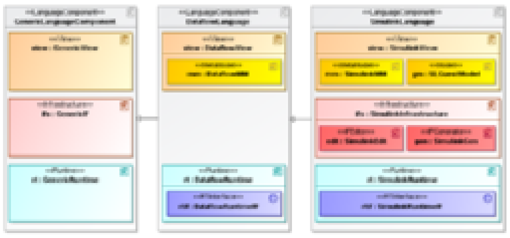


Figure 4: Language component specialization

This inheritance scheme supports language component hierarchies, e.g. the data flow hierarchy from the example. Specialized language components may introduce new guest models and tool chains but still re-use meta models of parent language components. Guest and meta model properties that are qualified with the type of their defining language component retain their type specifier; derived meta model types or replaced guest models therefore appear with their original qualifier. Therefore, meta models may only be extended through specialization and therefore are downward compatible to meta models of base components. Guest model access restrictions ensure that only transformations and infrastructure of one language component type may access the same guest model, and therefore also ensure that specialization does not lead to type conflicts.

3.1.3 *Conflicting view types*

The special relation *conflictingView* may be defined for any pair of views that must never be used together on one component. Specialized view types inherit this property from their base views. This way, it is ensured that conflicting realization views are never used together to define one component. This is handy if two language components are not sufficiently synchronized, but enable definition of similar things. For example, both Simulink and ASCET views define (different) data flow models. In order to operate properly on the same component, both views need to synchronize their whole model into a common meta model. While this is possible with CompoSE through transformations, this is impractical in real world applications. For this reason, both view types could be marked as conflicting instead, preventing developers to use them together on the same component.

3.1.4 *Checks*

Automated checks are executed similar to transformations every time when a connected property was modified. In contrast to transformations that produce output models, checks validate predefined properties or consistency rules. Typical application areas for automated checks in the CompoSE framework are DSL specific consistency checks across views. Similar to transformations, checks are currently developed in Java; for subsequent implementations, we plan the development of a DSL for specifying *OpenArchitectureWare* (OAW) based checks and model transformations using OAW's extend language.

3.2 CompoSE components

CompoSE components represent all system components – in this paper, we focus on the definition of software components though. Components are defined through properties – property values are modified through views. CompoSE supports two basic relations between components that are known from the UML: Component aggregation and component specialization. Currently, no distinction between aggregation and composition is made in CompoSE. However, due to the view concept, the behavior of both principles needs to be adapted.

Component aggregation is supported through the basic view type *AggregationView*. Aggregated components, i.e. components that consist of other components, are created through component aggregation only; no other non-aggregation realization views may be assigned to that component type. The two view types *AggregationView* and *NonAggregationView*, from which all non-aggregating view types derive are therefore marked as conflicting views. The realization of aggregated components is therefore only defined through the aggregation view – no other realization views may be applied to that component. Component specifications are not affected by aggregation views. Therefore, specifications of aggregated components are defined through specification views similar to any other view type. The aggregation view of CompoSE is similar to the composite structure diagram of UML that defines component substructures through instances, ports, and links. Figure 5 illustrates an example component aggregation. In the example, the component *CruiseControlSystem* is aggregated out of one instance of the component type *Filter* and one instance of the component type *Control*.

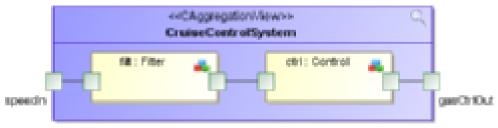


Figure 5: Aggregation view

Component specialization is more complicated than component aggregation, because it affects both component specification and realization views with unknown language semantics. Component specifications and realizations are defined through properties. Specialized components initially inherit all properties of their derived components. In addition, specialized components also may override properties of their base components, and therefore replace their value. This must be done through a compatible view that is able to modify the property in question. However, when replacing property values, the following additional restrictions apply, which are specific to guest languages and therefore are validated automatically by checks (cf. Section 3.1.4).

- Properties defining component specifications may not be lowered by specialized components - everything that was defined by the specification of the parent component must still be part of the specification provided by derived components.
- Properties defining realizations may be overridden and changed by specialized components as long as the component specification, and therefore inherited component specifications are met.

Depending on the guest language, language specific specialization constructs may be available. For example SDL and UML languages provide such concepts, while Simulink does not support type inheritance. If specialization constructs are available in a guest language, these constructs may be used for creating specialized guest models based on their parent guest models from parent components. Figure 6 illustrates this with an example.

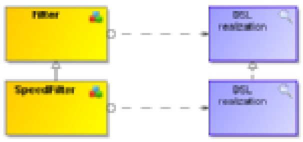


Figure 6: CompoSE component specialization

The example from Figure 6 illustrates CompoSE component specialization. The *SpeedFilter* component specializes the more generic filter component type. It also replaces the DSL based component realization. DSL dependent specialization constructs support specialization of the original realization of the base component. CompoSE aggregation and specialization are intentionally not strict and enforcing, because this would limit the applicability of CompoSE to a smaller number of guest modeling languages. CompoSE defines a necessary and sufficient set of constraints for aggregation and specialization that enable creation components and views with defined semantics.

4. MULFI-FORMALISM COUPLING

Up to now, we defined the integration of language components and therefore new modeling languages into Compose. View types define properties, meta models, and therefore containers for storing information. Transformations map models from one meta model into another, and bridge between guest models. Views also provide means to modify model elements by integrating infrastructure parts that represent existing language infrastructure, e.g. editors. Runtime adapters provide an interface between code generated by different language infrastructures, i.e. code generators, and bridge generated code. This works well as long as languages with conforming runtime semantics are coupled – for example Simulink and ASCET provide similar semantics, and therefore coupling is simple. However, multi formalism development requires the combination of modeling languages that implement different semantics. Bridging these languages is a challenging task, and the approach used for providing this coupling is an important design decision.

CompoSE supports this bridging through the *InterfaceView* and the *CouplingView* view types. The *InterfaceView* predefines port types that represent different semantics. The *CouplingView* supports connections between ports that represent similar semantics, or between port types for which a semantic mapping is defined. Basic port types that are defined by the *InterfaceView* are the following:

- Data flow ports (*FlowPort*) represent data flow semantics.
- Event ports (*EventPort*) represent asynchronously transmitted and received events.
- Control flow ports (ControlFlowPort) represent control flow semantics, e.g. the definition of operations with entry points and control flow transfer upon invocation.

The coupling of components that provide interfaces based on these port types, and therefore implement conforming semantics is defined through the coupling view. While coupling of compatible interfaces is trivial, the coupling view also defines coupling semantics that map from one interface type to another. Runtime adapters need to support these predefined mappings in order to support semantic coupling of their supported modeling languages. The following automated mappings are currently supported by CompoSE:

- Output data flow ports map to event ports by generating an event each time the output value changes. Event ports are mapped to input data flow ports by changing the respective data value each time an event is received.
- Control flow ports map to output data flow ports, if they define one operation with one parameter only that is then invoked upon parameter change. Mapping of control flow ports to input flow ports is supported if one operation is provided that carries one parameter, which will be the new value of the flow port.
- Mapping between event ports and control flow ports is supported as following: Whenever an event leaves an event port, which is connected to a control flow port, the corresponding operation will be invoked. If a response event is declared, the return value will be transmitted back upon the operation is completed. Operation execution is not synchronized with the execution of the component that transmitted the event. Mapping of control flow ports to realize required operation requires the definition of request/response pairs of events. The execution of required operations is mapped to the transmission of the request event. The calling component is suspended, until the corresponding response is required, in order to conform to control flow semantics.

By generating explicit adapter components using any supported language, more sophisticated mapping may be explicitly defined in addition to these predefined mappings. This coupling approach documents the basic rationale of compose to support black box components with defined white-box interfaces and properties; definition of component semantics are supported in a similar manner. While CompoSE is not aware of the complete semantic model of its black box components, it is aware of their interface semantics, and therefore is able to connect them to each other. Runtime semantics of components are supported in a similar way.

CompoSE predefines semantic models that are connected to views; not components. They are therefore stored in a property of the view type. These semantic models represent an abstraction of the runtime semantics of integrated modeling languages – since components may support multiple views (as long as restrictions regarding incompatible view types are not violated), they may as well implement different semantics. Predefined semantic models need to be supported by all runtime infrastructures. Additionally, new semantic models may be defined; the support of these models is then optional to runtime frameworks. The following semantic models are predefined:

- Data flow semantics realize a continuous data flow that continuously recalculates output values.
- Event based semantics provide semantics that realize views defining asynchronously executed behavior, which is triggered by events.
- Control flow semantics are passive – views using these semantics define component behavior that is triggered only through active transfer of control flow through control flow ports.

- Active control flow semantics are used to realize views which may receive control flow through control flow ports, but still provide an behavior on its own that is independent from explicit control flow transfer from other components.

Runtime frameworks, as already mentioned, need to implement at least these four semantic models. They also need to support components that apply different semantic models for different views.

5. COMPOSE APPLICATION EXAMPLE

In this section, we describe the integration of Simulink, an existing modeling language for data flow models. The Simulink language component supports the definition of component specifications and realizations through two different views, which consequently affect two different component properties. Simulink is supported by an industrial strength tool chain; this tool chain is integrated through the infrastructure interface into CompoSE.

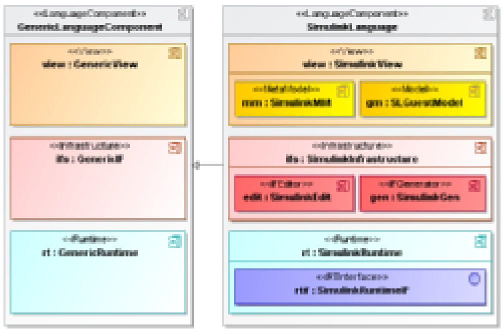


Figure 7: Simulink language component

Figure 7 describes the new language component *SimulinkLanguage*. New CompoSE language components extends directly or indirectly the type *GenericLanguageComponent*. The Simulink language component defines a white box meta model *SimulinkMM*, a guest model *SLGuestModel* that represents native .mdl files, and a model to model transformation that transforms parts of the native model into the white box model (not shown in Figure 7). Simulink views implement data flow semantics; their provided infrastructure is defined through an editor and a generator proxy. The definition of these infrastructure components is implementation specific. In our case, CompoSE was implemented into MagicDraw, which is a generic UML modeling tool. Infrastructure components are magic draw plugins that integrate code generation capabilities by calling code generators of integrated tool chains, or by invoking editors of generated tool chains. The editor connects to the Simulink editor, which is also part of the commercial tool chain. It is invoked in a similar manner as one if MagicDraws native UML diagram editors; however, changes in the model are synchronized only after saving in our implementation. Simulink diagrams are stored together with its native model representation in the guest model; after saving a diagram, transformations are invoked that extract relevant data from the diagrams and update component properties and white box meta models. The runtime interface of generated code is defined by the *SimulinkRuntimeIF* interface. Native Simulink code does not conform to that interface. Therefore, adapter code is generated by the *SimulinkGen* infrastructure proxy to mediate between the CompoSE runtime and generated Simulink code.

6. RELATED WORK

The author of [1] proposes a generic, component based framework for the evaluation of quality attributes like timeliness and safety. Each component gets four artifact types assigned: an encapsulated evaluation model, an operational/usage profile, composition algorithms, and evaluation algorithms. Based on these artifacts, a process for the evaluation of quality attributes is defined. This approach focuses clearly on evaluation of white box models; in contrast, CompoSE focuses currently on the efficient integration of new modeling languages as black box models, as well as providing an extensible framework for synchronizing information contained in different black box models.

The work presented in [2] present BIP, a component based development approach that supports multi formalism development of behavior components. BIP defines three layers per component, focusing on component behavior, interaction, and execution. The authors focus on behavioral realizations, and provide a framework for modeling components, as well as for the generation of glue code to link these components together at runtime. This approach focuses on correctness by construction and the adherence to properties while composing components, e.g. ensuring deadlock freedom. This is done via one common modeling language that component behavior is mapped to; in contrast, CompoSE provides the ability of integrating any language as language component. BIP could be used as a backend framework for performing formal analysis by defining the language of BIP as a white box meta model, and by providing transformations for guest models of language components into the formal language of BIP.

The authors of [3] present a framework for multi language development of embedded systems, which provides tool and model integration. Modeling languages are integrated into the proposed framework through adaption layers that provide a link between domain specific models and the common framework. Like CompoSE, connected models are divided into public parts that are exposed to other models, and private parts that are stored in a common repository, but will not be exposed. In contrast to this work, the basic framework described in [3] implements multi-language development on the tool level, not on modeling level. Therefore, no modeling language for the combination of multiple modeling languages is defined.

The authors of [4] provide an approach for multi formalism development that is much more tightly integrated than CompoSE; the described approach aims at integrating the meta models of used modeling languages. This is one difference between CompoSE and the approach presented in [4]: while CompoSE uses a central system model to synchronize modeling elements, the authors of [4] directly synchronize meta models with each other. While this approach is certainly appealing, the creation of the proposed consistency checking and mapping meta models costs considerable effort when defining a multi-language modeling approach for multiple already existing modeling languages. Depending on the amount of exported data and the complexity of the synchronization rules in views, CompoSE might provide such a tight synchronization as well. However CompoSE scales; in most cases full meta model synchronization is not required. Therefore, CompoSE will only synchronize a small subset of model data, resulting in a smaller and therefore less expensive synchronization approach in these situation (regarding both money and required processing power).

Ptolemy, which is presented in [5], is a famous approach for the application of execution semantics in Java environments, as well as for their evaluation, simulation, and composition. The focus of Ptolemy is on the semantic coupling, and simulation of components that implement different execution semantics. However, other aspects besides runtime semantics, e.g. the integration of modeling languages, tools, and (meta) model synchronization is not covered. Therefore, both approaches, Ptolemy and ComposE have a slight overlapping, which is the coupling of semantics. This coupling is currently in Ptolemy much more developed than in CompoSE, which focuses on the integration of modeling languages and light-weight model synchronization.

The authors of [6] describe Metropolis, which is a component based modeling framework, which is based on the following core concept of separation between communication and computation, and separation of functionality and architecture. Metropolis provides a common meta model that most existing models of computation may be transformed into. The metropolis model of computation is based on concurrent execution of action sequences; actions are subdivided into communication and computation actions. The main difference between CompoSE and Metropolis is its focus: CompoSE is an approach that aims at integrating (domain specific) languages, infrastructure and runtime frameworks in a light-weight manner. Runtime frameworks are combined using common runtime interfaces – as long as a runtime adapter and semantic mappings are provided, a specific language may be integrated into CompoSE. Metropolis provides a common model of computation that languages are transformed into. This requires a much more tight integration with respect to runtime models, and therefore much more integration effort.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented CompoSE, our multi-formalism modeling framework. CompoSE has been devised by applying principles from component based software engineering to the creation of a multi formalism modeling approach. It supports multi formalism development at three levels: the modeling level, the infrastructure level, and the runtime level. The principle for multi-formalism development with CompoSE is the application of guest models and guest languages that are plugged into one host model as language components. Views provide access to guest languages at modeling level and present data stored in models. Guest models are stored in their native file format and meta model of the guest language, meta models may be used to export whole guest models or part of it into the host model, so that other language components may access this information. This transformation is performed though explicit transformations. The CompoSE approach is different from most other approaches, because it provides a light weight language integration; the degree of language integration depends on provided meta models and transformations, and may therefore be adapted. This is an important aspect for its practical applicability, where integration effort equals to money.

Through the concept of guest models, existing languages, infrastructure, and runtime frameworks may be used with CompoSE. This is especially important in industry, because multi formalism approaches are only accepted if they support established and well proven tool chains. The separation between black box guest models and white box meta models enables a rapid integration of new modeling languages, because only relevant attributes of guest models need to be synchronized with the host model; full meta model synchronization is possible, but not necessary with CompoSE. We have proven the applicability of CompoSE through the integration of the existing Simulink language as language component.

Ongoing and future work with respect to CompoSE is the definition of a set of views for systems modeling in the automotive industry. Additionally, the definition of formal semantics for CompoSE language constructs, relations, as well as for language and formalism coupling is currently ongoing work. Once this is finished, clear coupling semantics will be available, as well as an approach for the integration of new coupling semantics.

8. REFERENCES

[1] L. Grunske, *Early Quality Prediction of Component-Based Systems - A Generic Framework*, Journal of Systems and Software, Elsevier, Volume 80, Issue 5, May 2007, pp. 678-686

[2] G. Gössler, J. Sifakis, *Composition for Component-Based Modeling*, Science of Computer Programming, Volume 55(1-3), 2005

[3] J. El-Khoury, O. Redell, M. Törngren, *A Tool Integration Platform for Multi-Disciplinary Development*, Proceedings of the 2005 31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'05), Porto, Portugal, 2005

[4] S. Burmester, H. Giese, J. Niere, M. Tichy, J. P. Wadsack, R. Wagner, L. Wendehals, A. Zündorf, Tool Integration at the Meta-Model Level within the FUJABA Tool Suite, Proceedings of the Workshop on Tool-Integration in System Development (TIS), 2003

[5] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong: Taming Heterogeneity - the Ptolemy Approach. Proceedings of the IEEE, v.91, No. 2, January 2003.

[6] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe. Modeling and Designing Heterogeneous Systems, volume 2549 of LNCS, pages 228–273. Springer-Verlag, 2002.

ITML: A Domain-Specific Modeling Language for Supporting Business Driven IT Management

Ulrich Frank
ulrich.frank@uni-due.de

David Heise
david.heise@uni-due.de

Heiko Kattenstroth
heiko.kattenstroth@uni-due.de

Chair of Information Systems and Enterprise Modeling
University of Duisburg-Essen
Universitaetsstr. 9, 45141 Essen, Germany

Donald F. Ferguson[a]
donald.ferguson@ca.com

Ethan Hadar[b]
ethan.hadar@ca.com

Marvin G. Waschke[c]
marvin.waschke@ca.com

CA Inc.
[a]New York, NY, USA; [b]Herzelia, Israel; [c]Washington, NY, USA

ABSTRACT

Management of today's IT is a challenging task that requires a profound understanding of both the IT landscape and the relevant business context. Numerous relations and dependencies between business and IT exist, which have to be accounted for, e.g., for better IT/business alignment. This paper presents ITML (IT domain specific Modeling Language) integrated with a comprehensive method for enterprise modeling. The language advantages are illustrated in terms of support for profound analyses, development of sophisticated IT Management tools (build-time), and use of corresponding models at run-time, e.g., as part of IT Dashboards.

Keywords

Domain-Specific Modeling Language, Enterprise Modeling, IT-Management

1. MOTIVATION AND SCOPE

IT Management is confronted with remarkable challenges. On the one hand, it is expected to serve the business with high efficiency, on the other hand it must cope with the diversity of IT platforms, networks, and information systems and their interdependencies. From a technical perspective, there is need for integrating and consistently maintaining these IT artifacts. From a managerial perspective, the transition from taylorism to process-oriented organizations as well as the growing relevance of cross-organizational business processes emphasizes the need for information systems that are not restricted to particular business functions, but that provide effective and versatile support for business processes and fulfill the business' needs ('IT/business Alignment', cf.[10, 15]). These challenges are made more difficult by language barriers between IT and business and between the different IT domains.

To cope with these challenges, methods and tools are required that support the range of IT management tasks. Existing tools and methods for IT Management are unsatisfactory in this respect. Approaches for integrating IS or managing the IT infrastructure, such as Enterprise Application Integration and Middleware (e.g., [20]) or Configuration Management Databases (CMDBs; e.g., [19]), focus on issues such as hardware and its operational metrics, e.g., address the matter of physical data exchange or management of concrete IT resources and the implementation of IT processes. Their support for elaborate technical analyses, e.g., for checking IT architectural weaknesses, or for integrating heterogeneous systems (cf. [6]) is somewhat limited, since these approaches abstract away the business context of the IS. In contrast, IT Management frameworks such as ITIL[1] or CobIT[2] present high-level guidelines for IT organization's services and processes. They provide an abstracted approach for managing IT for typical IT processes, and occasionally define metrics and key performance indicators for evaluating the quality of the operational status of the IT domains. However, there remains a gap between the IT Management and the business context on the one hand, and the detailed technical level on the other. While the gap is supposed to be overcome by IT managers, the complexity of this task suggests appropriate support – both for analysis purposes and for communicating with various stakeholders. In this respect, the motivation is twofold: First, we propose a domain-specific modeling language (DSML) for modeling IT infrastructures – the IT Modeling Language (ITML). It provides concepts for conveniently creating illustrative and consistent models of IT infrastructures, which enable various types of analyses and transformations. At the same time, it can be supplemented by corresponding process models to engineer modeling methods for IT Management. Second, the ITML is intended to support the design of tools for IT Management (build-time). We also show that ITML is useful as a versatile management instrument at run-time of these tools, e.g., by using diagrams as front end for instance data.

The ITML is part of a comprehensive method for enterprise modeling that includes various other modeling languages, e.g., an organization modeling language or a strategy modeling language. Therefore, ITML models can be supplemented by models of the relevant context in order to promote IT/business alignment and foster communication between stakeholders with different professional backgrounds.

[1]IT Infrastructure Library, [19]
[2]Control Objectives for Information and Related Technology, [11]

The remainder of the paper is structured as follows: In Section 2, the requirements for a DSML for IT Management are analyzed. In Section 3, two exemplary use cases illustrate concepts and graphical notation of the ITML and the language's benefits for IT Management. Subsequently, the conceptual background of the ITML is presented in the form of a meta-model and a language architecture. Related work is discussed in Section 5. The paper closes with an evaluation of the solution and an outlook on future work.

2. REQUIREMENTS

The following requirements analysis is aimed at preparing a foundation for the design of the ITML, and clarifies the choice between a domain-specific modeling language in general and a general-purpose modeling language (GPML).

For many planning and analyses scenarios, accounting for all resources in all details is not necessary. In fact, too much detail can obscure goals and make planning and analysis more difficult than an intelligently simplified view.

Req. 1 – Reduction of Complexity: IT Management demands for abstractions that allow for focusing on those concepts that are pivotal for certain types of analyses and application scenarios. This requires avoiding distraction caused by irrelevant technical detail. Nevertheless, ignoring technical details on principle will not be satisfactory, since some scenarios require information about concrete instances.

Ever changing and evolving technologies and corresponding "buzz words" are distinctive of the IT domain – although the basic concepts seldom change.

Req. 2 – Protection of Investment: To protect investments into models, the language concepts should neither represent technical aspects that are subject to change nor features that are specific to particular products. Note that stressing this kind of abstraction also contributes to the protection of investments into the IT itself, since it makes IT infrastructures less vulnerable against changes of variable details.

IT Management is hampered because various stakeholders, such as end-users, executives, IT experts etc., need to be involved in planning, designing, and managing IT. The language barriers between these groups may cause misunderstanding, compromising the efficiency of IT systems.

Req. 3 – Support for Multiple Perspectives: On the one hand, meaningful representations of IT at different levels of abstraction are required to satisfy the needs of the various groups of prospective users of the language. If possible, they should correspond to concepts and representations current in the prospective users' domain. On the other hand, these perspectives should be integrated to foster communication between stakeholders with different professional backgrounds.

IT is not an end in itself. Instead, it supports an organization's business processes, enterprise goals, and – in general – its competitiveness. Hence, adequate management of IT requires a profound understanding of the interdependencies between business and IT.

Req. 4 – Business Context: IT Management must not be treated as an isolated function. Instead, users should be informed of the organizational context of IT. This requires including concepts that represent the business context, e.g., strategies and goals or business processes.

IT Management still strives to integrate the multitude of different tools that are scattered over the enterprise. Data about the enterprise's IT areas (e.g., hardware, software, IT services, security, governance) are often gathered and managed separately in different information systems. This leads to independent data silos, which jeopardize data consistency (cf. [6]).

Req. 5 – Integration: To support analyses on interoperatibilty of IT systems, there is need for concepts to express data or functional similarities or functions and integration deficiencies within business processes. To develop conceptual foundations for integrating heterogeneous artifacts, there is need for concepts – i.e., meta types – that can be instantiated into a range of corresponding types, e.g., different implementation of a type "Customer".

Models created with the ITML should be used to design tools for IT Management (build-time), for instance, by transforming the IT models into an enterprise-specific database schema for software to manage instances of hardware; and for providing versatile and extensible operational interfaces that can be used during business operation (run-time).

Req. 6 – Formal Foundation: The semantics of the ITML should be specified precisely enough for unambiguous transformations into implementation documents such as code.

Ostensibly, general-purpose modeling languages address the requirements. One could argue that a GPML like the 'Unified Modeling Language' (UML, [18]) or the 'Entity Relationship Models' (ERM, [2]) meet these requirements. However, such an approach has serious deficiencies (e.g., [3, 13, 16]). First, a GPML does not effectively support the construction of domain-specific models, because its syntax and semantics are designed to express any model; they are not designed to exclude inconsistent models, and thus they do not constrain users from creating – from a domain's perspective – wrong models, e.g., with 'hardware running on software' (*lack of integrity*). Second, it would be rather inconvenient to describe IT resources using only generic concepts such as 'Class', 'Attribute', or 'Association' (*lack of convenience*), which are well-suited to modeling software, but were not chosen with modeling IT in mind. Third, the graphical notation, i.e., concrete syntax of a GPML does not contribute to an illustrative visualization in the graphic idiom of IT stakeholders, such as business managers (*lack of comprehensive models*); hence, it would, if at all, provide only little support for cross-IT domain communication.

Against this background, a DSML specific for IT Management seems to be more likely to meet the preceding requirements than a GPML. In addition to the general requirements presented above, the design of a DSML is guided by the objective to reconstruct existing technical languages, in this case the professional language of IT Management. This is for two reasons. First, it benefits from the elaborate and proven technical language of the domain instead of reinventing the wheel. Second, it makes the DSML more comprehensible, since its concepts correspond closely to terms the prospective users are familiar with – i.e., provide high semantics (cf. [6]).

3. ILLUSTRATION OF THE SOLUTION

The following scenario serves to illustrate the use of the ITML to support IT Management. The scenario is divided into two steps. First, focus is on a model of IT infrastructure that is integrated with a business process model. This step aims at indicating the potential of the modeling language to support for planning as well as 'strategic' analysis. Second, the models are extended with instance level data, which promises to support decisions dealing with concrete IT resources, e.g., a particular server or software. This requires integrating an ITML modeling tool with corresponding systems at the operational level such as CMDBs.

Figure 1 presents a model of an IT landscape (Resource/ Service and Location level), which is extended by a business perspective (Process Map). It shows various types of IT concepts such as (from top to bottom) IT services, software, diverse hardware like database systems, mainframe, mail servers, or web server, and locations (data centers). Furthermore, the elements are interrelated if they are dependent in some way, e.g., software runs on hardware and enables IT services. Such a model of the enterprise's IT already enables various analyses for IT Management. Two simplified examples are illustrated below.

Example 1 – Outsourcing: The depicted model at type level provides a foundation for outsourcing decisions. First, depending on the interencies (coupling) an IT resource type – e.g., the hardware type 'DBS 3' – has with other IT resource types, it might be a good/poor candidate for outsourcing. This is based on the following assumption: The higher the amount of interdependencies, the more complicated it will be to detach it and outsource to an external location. However, accounting only for the sheer amount of interdependencies would be an oversimplification. Rather, it is necessary to evaluate the importance of the interdependencies, i.e., the associations of a resource type. For instance, one can use the depicted models to assess the dependency between 'DBS 3' and the software type 'SAP BW'. If the association between these two types does not indicate strong coupling, and decoupling might be accomplished relatively easy, the impact of outsourcing will be less substantial.

With respect to the business perspective, the models allow for evaluating a resource's relevance for the business, e.g., by analyzing the 'business impact' of a resource in case of its breakdown/malfunction. In our example, an analysis of the associations among the modeled types reveals that the 'DBS 3' is used – to a yet unknown extent – in two services ('customer rating', 'customer contact'), which in turn support various business process types. Hence, in contrast to predominant descriptions of IT – such as records in configuration databases (e.g., a CMDB) – the model-based description, although still on type level, already indicates manifold advantages, like comprehensive analyses. The illustra-

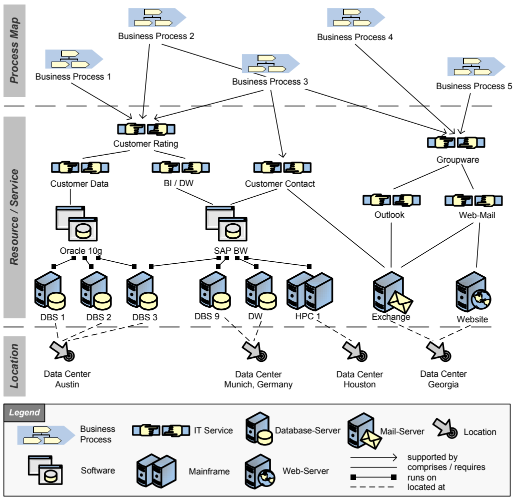


Figure 1: Exemplary Scenario

tive visualization further allows for inspecting a model on sight and by stakeholders that are not familiar with, e.g., data-querying languages that are necessary for analyses in database-oriented apporaches like CMDBs.

In a second step, these models can be enriched with additional information about the actual instances. Figure 2 shows the IT/business process models, in which two types, the 'business process 2' and the 'customer data' IT service, are enhanced with information about current instances. This requires that the types in the model (e.g., the IT service type 'Customer Data') are 'linked' to corresponding instances (e.g., information about actual instance of this service). Such an integration of models with corresponding instance information, i.e., the use of models at run-time, fosters a more profound decision-making and a variety of analyses than analyzing at instance level only, since information about particular instances are now enriched with the business context, while at the same time distracting complexity of the domain is still reduced. For our example, this can be applied to:

- Resource type 'DBS 3': Is there need for an upgrade in any way (e.g., based on purchase date, end of maintenance contract, number of breakdowns/incidents, costs)?
- Software types and their utilization of resource types: How frequent ly do they depend on each other (e.g., based on capacity utilization, amount of database accesses)?
- IT services: How frequent are the services accessed (e.g., charges, amount of instances)?
- Evaluating the return on investment by monitoring the IT services usage: How frequent are the services accessed (e.g., charges, amount of instances)?
- Business processes adjustments: What is the process' criticality (e.g., based on its value to customer, revenue, amount of instances)?

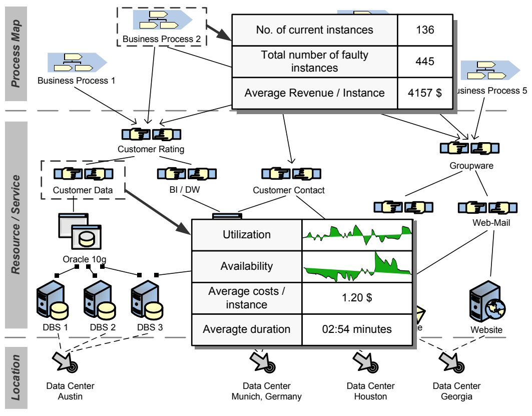


Figure 2: Exemplary Scenario enhanced with instance information

Example 2 – Consolidation/Integration: Already the development of an ITML model helps to structure the domain of interest and identify potential similarities, for instance between services offered to the business processes. Thereby, such models foster identification of candidates for consolidation and integration, e.g., of redundant data centers caused by mergers and acquisitions. If a data center offers services that are identical or closely related to services from another data center, it might be a candidate for consolidation. In many cases, such analyses still require an inspection and interpretation of the models by the users. However, depending on the analyses and application scenario tool-support might be possible, e.g., by highlighting IT services that have similar relationships; and, for instance, in contrast to querying datasets in a CMDB, it is more intuitive and comprehensive in terms of *Req. 3 & 4*. In the example illustrated in Fig. 1, the data center 'Austin' offers the service 'Customer Data' that is apparently closely related to the service 'Customer Contact' provided by data center 'Munich'. Moreover, both data centers jointly provide the service 'Customer Rating'. In order to decide about consolidating similar or related services into a single data center, the models can be enriched with information about the instances, for example, to assess the importance of the different services and accordingly of the data centers, the criticality of the underling infrastructure, and type of solution, the number of problems and tickets associated with the instance level over time, and more. Note that in decision-making usually far more information than only name and associations of, e.g., an IT service type – such as depicted in Fig. 1 & 2 – is required; in this respect, the models presented are simplifications (i.e., attributes are omitted for sake of space restrictions).

4. CONCEPTUAL BACKGROUND: META MODEL & LANGUAGE ARCHITECTURE

The DSML is specified in a meta model using the Meta Modeling Language MML (cf. [7]), which was specifically designed for specifying languages for enterprise modeling that feature a high degree of inter-language integration. The design of the DSML is guided by several objectives, driven by the requirements identified in Section 2. First, the modeling language should provide concepts that represent a reconstruction of the technical terminology of the IT domain (cf. *Req. 1*). This requires finding abstractions that closely correspond to concepts in the domain – i.e., provide a high level of semantics – in order to facilitate a comfortable use of the DSML and communication between the involved stakeholders. At the same time, the concepts of the modeling language should be rather generic in the sense that they apply to a wide range of enterprise settings and over a longer period (cf. *Req. 2*). The reconstruction also pertains to business terminology. The language has to consider concepts from the business domain that might be relevant for IT Management and provide integration between both domains (cf. *Req. 4*).

While there are various ways to structure the IT Management domain, we follow Kirchner [14], who proposes three categories of concepts for an earlier version of the ITML: *technological concepts*, such as hardware, software, network, peripherals, and so on; *organizational concepts*, which include business processes, roles/people, costs, and goals; and *additional abstractions* like IT services or information systems. Figure 3 illustrates a semantic net of the basic relations of the most prominent core concepts (cf. [14]): 'hardware' is located at a 'location' and required by 'software'. An 'information system' is an abstraction over a certain set of software and hardware. It provides 'IT services', which support 'business processes' and, in the end, contribute to the realization of the company goals and strategies. Organizational roles are related to these concepts in various ways, e.g., by means of utilization, maintenance, or responsibility, and as part of information systems (e.g., a database admin) or apart from them (e.g., help desk staff). Consequently, these core concepts constitute the foundation for the ITML language specification presented in Section 4.1.

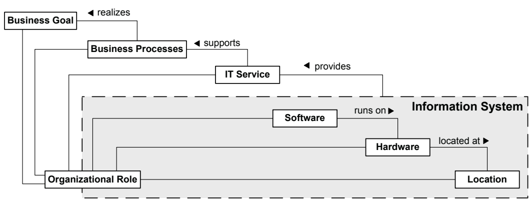


Figure 3: Core Concepts of ITML

A second design objective refers to offering a graphical notation (concrete syntax). In contrast to, e.g., textual or formal descriptions, a graphical notation supports a rich and intuitive documentation while it at the same time can depict numerous relations in a more comprehensible way. The notation has already been illustrated to some extent Section 3.

4.1 The ITML Meta Model (Excerpt)

The concepts in Figure 3 already provide a basis for the ITML meta-model. However, the specification of the modeling language still faces a number of challenges. The three most pivotal ones are discussed below. Subsequently, corresponding design decisions are presented.

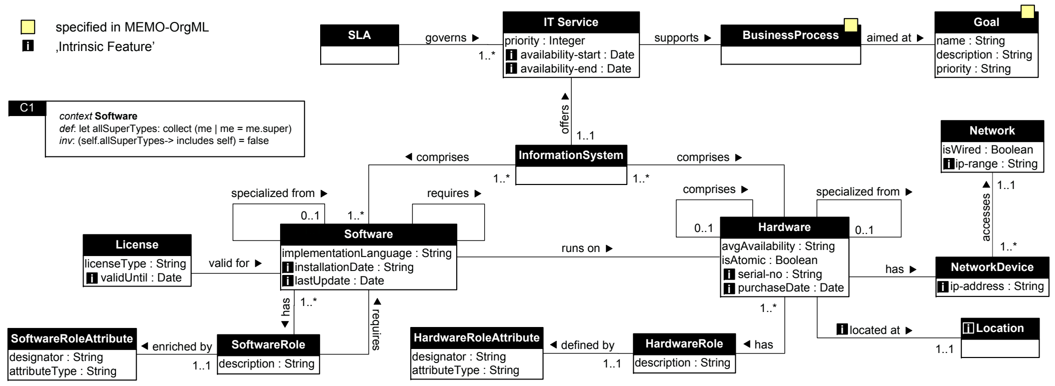


Figure 4: The ITML Meta Model (Excerpt)

Modelling Challenge A: Contingent Classification. Besides generic concepts such as 'software' or 'hardware' there exists a plethora of further refinements and characterizations for these concepts. For instance, software can be categorized with regard to its architecture (e.g., client/server), primary purpose (e.g., database management, middleware, web server), or its nature (e.g., infrastructure, application, frontend). From a modeling perspective such differentiations can be realized in different ways: In terms of specific meta types, by the use of generalization/specialization (i.e., differentiations as sub-types of 'software'), as a value of an attribute of the generic meta type 'software' (e.g., an enumeration 'type of software'), or as a role. At first glance, reconstructing all classifications as separate meta types or sub-types would conform to *Req. 1*. However, such a reconstruction would not be compliant with the demand for invariant concepts (*Req. 2*) and to an unambiguous assignment of real-world entities to concepts of the language. The concerns can be that the classification of software is often superficial and a matter of perspective – i.e., while in one decision scenario a software might count infrastructural, it can be regarded as application software in another. Though accurate in terms of technical concerns of a repeating structure, it remains conceptually different in a business context. Prominent examples are modern operating systems (usually infrastructure software) that provide integrated functionalities that count as application software; or complex application servers that provide, among others, database, middleware, and server functionalities. Furthermore, software can be assigned to several categories. The same accounts for the concept of 'hardware'. Even more, consider the convergence of devices, when multi-purpose hardware solutions such as a combined print/fax/copy machine, or a media-phone-handheld computer are introduced, classification issues are more evident

Modelling Challenge B: Interfacing to Instance Level. The DSML is designed for creating models at type level (cf. *Req. 5*); hence, the concepts in the models represent types. For design purposes, this focus is usually sufficient and necessary at the same time. However, often, it is required to differentiate between types and instances (cf. *Req. 1*). Ignoring instance information in general might generate wrongful assumptions as indicated by the above application scenarios. Therefore, the language concepts should allow for referring to instances somehow.

Modelling Challenge C: Type Differentiation. The modeling challenge pertains to the restrictions given by the type/instance dichotomy commonly applied in conceptual modeling (such as in [17]) and the semantic differences between instantiation and specialization. A discrimination of types and instances is – especially in the IT domain – not trivial, and it remains often unclear whether a real-world entity is represented as a modeling concept (i.e., a type) or as an application (i.e., an instance) of a modeling concept. Take, for instance, the meta-type 'software'. Possible type instantiations could be 'Word Processing Software', 'Microsoft Word', 'Microsoft Word 2003', or 'Microsoft Word 2003 Business Edition'. However, at the same time, 'Microsoft Word' could be regarded as an instance of a meta-type 'Word Processing Software' or as a specialization. Hardware concepts raise similar abstraction problems. For example, 'Printer' could be conceptualized as a meta-type with instantiated types such as 'Laser Printer' or 'Ink Jet Printer'. Alternatively, 'Laser Printer' could be specified as meta-type, with 'Color Laser Printer', 'HP Laser Printer XY-Series' etc. as instantiated types (cf. [5]). The decision for a certain abstraction, i.e., what is regarded as software type, as its specialisation and as its instance, variies among enterprises. If a modeling language is not flexible in this regard, it might constrain its application range or even be unsuitable for enterprises. Hence, the ITML should provide users with appropiate concepts and guidelines.

Figure 4 illustrates an excerpt of the ITML meta-model. Note that certain aspects were simplified due to space restrictions. It also shows only one exemplary OCL-constraint (*C1*), as well as '0..*'-cardinalities are omitted for reason of clarity. The meta-model illustrates the design decisions which target the above challenges:

Ad A – 'Roles': To enable users to express that a software/hardware type can be assigned to different categories, we use the concept ***'role'***. Software and hardware types are instantiated from meta-types *software* or *hardware*. To assign a type a specific purpose, it can be associated to an according *softwareRole* or *hardwareRole*, which either already exists or has to be instantiated (cf. [14]). In order to reuse and extend software roles that also provide (higher) semantics, a software role can be enriched with further attributes (*SoftwareRoleAttribute*/ *HardwareRoleAttribute*), which allows for defining individual sets of software/hardware roles. While it would be possible to present a set of predefined role types by specializing *softwareRole*/*hardwareRole* in an enterprise specific language modification, our solution is more convenient because it can be used by users without meta modeling expertise – i.e., it is not necessary to adapt the meta model.

Ad B – 'Intrinsic features': There are certain apparent features of IT artifacts that we cannot express through the specification of a type only, since they are used to represent instance states (e.g., an IP address of a network device, serial number of hardware, or installation date of software). With regard to *Req. 1*, neglecting such instance features would not be satisfactory. To meet this challenge, we use the concept of 'intrinsic features' [7]. An intrinsic feature is a type, an attribute or an association that reflects a characteristic that we associate with a type that applies, however, only to the instance level. Hence, an intrinsic feature within a meta model is not instantiated at type level, but only one level further, i.e., at the instance level. In the MEMO Meta Modeling Language (MML), which is used to specify the present meta model in Fig. 4, intrinsic features are marked by an 'i' that is printed white on black (cf. [7]). A meta type that is marked as intrinsic, is actually a type (such as 'Location').

Ad C – Customized Specialisation: With respect to the restricted number of instantiation levels available for modeling, there is no perfect solution to this challenge. The ITML offers two approaches to cope with it: First, the meta-types are restricted to a few rather generic ones (such as 'Hardware', 'Computer', 'Printer' etc.– some are not shown in the excerpt). More specific types would then be created by instantiation, e.g. 'Laser Printer' from 'Printer'. Second, if there is need to create more specialized types, this can be done by making use of a 'specialized from'-relationship, which is specified for IT artifacts such as Hardware or Software. Note that the introduction of a specialization relationship implies additional constraints. These constraints are not included in the excerpt – along with further attributes and additional concepts such as (software) technical Standards, Software/Hardware Interfaces, and Organizational Roles.

4.2 Language Architecture

The integration with the business context requires to offer not only concepts that represent the IT domain, but that also account for concepts from business (cf. *Req. 4*). In the ITML meta-model the business context is represented by the meta types business process and goal. It would be inefficient to "re-invent" these modeling concepts for the ITML again, especially since they are not its primary concepts and main focus. To promote such reuse the ITML is integrated with other modeling languages for, e.g., business process or goal modeling in a way that allows for reusing concepts at the meta level and, by this, fosters the integrity of the corresponding models at the type level.

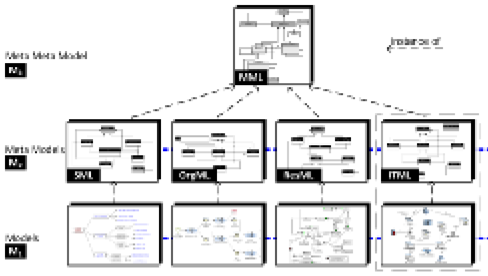


Figure 5: MEMO Architecture and the integration of ITML

For this purpose, the ITML is integrated with a method for enterprise modeling – the *multi-perspective enterprise modeling* (MEMO) method [4] – that already contains a number of domain-specific modeling languages. MEMO is multi-perspective in that it provides different groups of stakeholders with special abstractions and specific views on their relevant activities within the enterprise. Figure 5 illustrates a simplified version of the language architecture of MEMO. A more elaborate version can be found in [7]. All modeling languages within MEMO, including ITML, are specified using the MEMO Meta Modeling Language (MML, [7]) at the M_3 level. This fosters their integration since they are specified using the same modeling concepts – which allows for defining and re-using common concepts at the meta-level (M_2). This consequently leads to integrated models at type level (M_1), e.g., integrated IT and business process models. Thus, the ITML is integrated with a DSML for business processes and organizational structure (organizational modeling language, OrgML [4]), for resources (resource modeling language, ResML [12]), and for strategies and goals (strategy modeling language, SML [8]).

Concerning the use of the ITML, the integration with MEMO broadens the scope of the ITML as it is extended from an IT perspective to a more comprehensive view on an enterprise, thus fostering IT/business alignment and communication between the various enterprise stakeholders.

Note, even the excerpt in Fig. 4 might overstrain some users. Hence, the 'technical' details of modeling should be hidden from users, e.g., by a corresponding modeling tool. Furthermore, the amount of concepts that are necessary and the preferred level of detail vary between decision scenarios and the stakeholders involved. Thus, it is necessary to adapt the application of the language from case to case – which leads to the topic of 'method engineering' (cf. [1]). Method engineering is supported by MemoCenterNG[3], a modeling environment that implements the presented language architecture in Fig. 5). Thereby, it offers modeling editors for the MEMO languages, which includes an ITML modeling editor (see [7]). It also offers a meta-model editor that allows for creating further model editors. This enables experienced users to generate model editors that are based on the ITML meta-model and provide customized diagram types that, for

[3]Visit http://www.wi-inf.uni-due.de/fgfrank/memocenter-en or refer to [7] for more details.

instance, hide concepts that are irrelevant in the specific application scenario.

5. RELATED WORK

In practice, various tools for IT Management, e.g., for monitoring, network management, or IT Service Management are available, which are often based on a Configuration Management Database (CMDB) or similar databases. Such tools often allow for arbitrary models and do not provide a clear separation between different levels of abstraction, e.g., type and instance level (cf. *Req. 1–3*). Furthermore, these models mainly focus the management of instance data about IT resources and hardly account for the business context (cf. *Req. 4*).

An example of a related approach for modeling IT landscapes is the Common Information Model (CIM) published by the Distributed Management Task Force (DMTF[4]). It comprises a meta-model that defines basic concepts used for vendor-independent descriptions of IT landscapes. In this regard, the CIM solely focuses on describing IT resources and its main purpose is to use it as a schema for corresponding databases. However, since CIM does not provide any concepts from the business domain, e.g., abstractions for business processes, it neither contributes to a better IT/business alignment nor fosters communication between different stakeholders (cf. *Req. 3 & 4*).

Note, there are some modeling tools available (e.g., ARIS[5], ADOit[6]) that provide concepts for modeling IT landscapes and – to a certain extent – allow to integrate them with models of the business context. However, their language specification is usually not available. As far as we can extrapolate by examining these tools, they do not foster the development of customized tools, e.g., through code generation (cf. *Ref. 6*). Finally, these approaches and tools do not go beyond a company's boundaries – yet, there exist no mechanisms for exchanging and reusing IT models between enterprises or even within an enterprise (cf. *Req. 2 & 5*).

6. EVALUATION & FUTURE RESEARCH

In this paper, we outlined a domain-specific modeling language for IT landscapes. The language is aimed at accomplishing transparency by structuring and integrating the domain and, by this, reducing its complexity in order to support IT Management.

The language was designed to fulfill six requirements: The core concepts of the ITML have been reconstructed from the IT domain to provide abstraction that focus on relevant aspects (*Req. 1*). For this purpose, irrelevant technical details have been omitted. Thereby, focus is on invariant concepts so that efforts and investments made into IS/models are protected (*Req. 2*). This also fosters reuse of models and integration of IS (*Req. 5*). By embedding the ITML into a method for enterprise modeling, representations of IT infrastructures can be enriched with related representations. This supports not only accounting for the business context, e.g., for a better alignment with business objectives (*Req. 4*) but also facilitates the communication between stakeholders with different professional backgrounds (*Req. 3*). Finally, the semantics of the ITML allow for transforming an IT model into, e.g., a database schema for a CMDB, as well as for code generation in order to develop (integrated) software for managing IT resources (cf. [9, 13]; *Req. 6*).

Currently, the ITML primarily focuses on modeling IT infrastructures, albeit it also accounts for services and processes. Modeling of further important aspects of the IT domain, such as IT projects, is subject to future work. Moreover, we plan to refine the language specification, e.g., by further research projects with business practice, and advance the implementation of the modeling environment. This includes research on the aspects addressed, such as promoting the use of the ITML in IT Management dashboards, i.e., using models created with a DSML at run-time for advanced information systems. In this regard, the integration of instance information is a pivotal issue, which will be addressed next.

Compared to other approaches as described in Section 5, our approach shows clear advantages, mainly by featuring a higher level of semantics. Beyond satisfying the requirements discussed above, the ITML promises to supplement de-facto standards such as CobIT and ITIL by providing common concepts to describe the IT Management domain – thus fostering the integrated use of both standards. Further, the ITML serves as an instrument for bridging the gap between their high-level guidelines and the technical, i.e., more detailed view of IT Management. As highlighted in the application scenario and the language specification, the ITML features integration with the instance level, i.e., ITML models can be used to generate schemata for databases that manage instance data. Taken one-step further, this integration can be used to leverage ITML models for run-time, too. Envisioning an integration of the MEMO modeling environment with operational systems that manage the instance data (e.g., workflow management systems or the CMDB), ITML diagrams can also serve to build versatile and meaningful 'dashboards' for IT Management. Furthermore, we illustrated, that the ITML and corresponding models of the IT landscape foster comprehensive analysis and facilitate profound decision-making. Finally and with respect to *Req. 5*, the language serves as conceptual foundation for integrating information systems on a level of semantics that goes beyond all current capabilities: Information systems can describe themselves by referring to IT models extended (i.e., integrated) with business models – hence, enabling *self-referential information systems* (cf. [9]).

7. REFERENCES

[1] S. Brinkkemper. Method engineering: engineering of information systems development methods and tools. *INFORM SOFTWARE TECH*, 38(4):275–280, 1996.

[2] P. P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM T DATABASE SYST*, 1(1):9–36, 1976.

[3] R. Esser and J. Janneck. A Framework for Defining Domain-Specific Visual Languages. In *Proc. of the* 1st *OOPSLA Workshop on Domain-Specific Modeling (DSM'01)*, Tampa Bay, 2001.

[4] U. Frank. Multi-Perspective Enterprise Modeling

[4]http://www.dmtf.org/standards/cim/

[5]http://www.ids-scheer.com

[6]http://www.boc-group.com

(MEMO): Conceptual Framework and Modeling Languages. In *Proc. of the* 35^{th} *Hawaii International Conference on System Sciences (HICSS-35)*. Honolulu, 2002.

[5] U. Frank. Ebenen der Abstraktion und ihre Abbildung auf konzeptionelle Modelle – oder: Anmerkungen zur Semantik von Spezialisierungs- und Instanzierungsbeziehungen. *EMISA Forum*, 23(2):14–18, 2003.

[6] U. Frank. Integration – Reflections on a Pivotal Concept for Designing and Evaluating Information Systems. In R. Kaschek, C. Kop, C. Steinberger, and G. Fliedl, editors, *Information Systems and e-Business Technologies*, pages 111–122, Berlin, Heidelberg, 2008. Springer.

[7] U. Frank. The MEMO Meta Modelling Language (MML) and Language Architecture. ICB Research Report 24, Institute for Computer Science and Business Information Systems (ICB), University of Duisburg-Essen, 2008.

[8] U. Frank and C. Lange. E-MEMO: a method to support the development of customized electronic commerce systems. *Inf. Syst. E-Business Management*, 5(2):93–116, 2007.

[9] U. Frank and S. Strecker. Beyond ERP Systems: An Outline of Self-Referential Enterprise Systems. ICB Research Report 31, Institute for Computer Science and Business Information Systems (ICB), University of Duisburg-Essen, 2009.

[10] J. C. Henderson and N. Venkatraman. Strategic alignment: Leveraging information technology for transforming organisations. *IBM SYST J*, 32(1):4–16, 1993.

[11] IT Governance Institute, editor. *CobiT 4.1: Framework, Control Objectives, Management Guidelines, Maturity Models.* IT Governance Institute, Rolling Meadows, 2007.

[12] J. Jung. *Entwurf einer Sprache für die Modellierung von Ressourcen im Kontext der Geschäftsprozessmodellierung.* Logos, Berlin, 2007.

[13] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley, New York, 2008.

[14] L. Kirchner. *Eine Methode zur Unterstützung des IT–Managements im Rahmen der Unternehmensmodellierung.* Logos, Berlin, 2008.

[15] J. N. Luftman, P. R. Lewis, and S. H. Oldach. Transforming the Enterprise: The Alignment of Business and Information Technology Strategies. *IBM SYST J*, 32(1):198–221, 1993.

[16] J. Luoma, S. Kelly, and J.-P. Tolvanen. Defining Domain-Specific Modeling Languages: Collected Experiences. In *Proc. of the* 4^{th} *OOPSLA Workshop on Domain-Specific Modeling (DSM'04)*, Oct 2004.

[17] Object Management Group. Meta Object Facility (MOF) Core Specification, http://www.omg.org/docs/formal/06-01-01.pdf. 2009-08-07.

[18] Object Management Group. Unified Modeling Language Infrastructure, http://www.omg.org/docs/formal/07-11-04.pdf. 2009-08-07.

[19] Office of Government Commerce, editor. *ITIL – Service Operation.* The Stationery Office, London, 2007.

[20] D. Serain. *Middleware and Enterprise Application Integration.* Springer, London, 2002.

Domain Specific Languages for Business Process Management: a Case Study

Janis Barzdins Karlis Cerans Mikus Grasmanis
Audris Kalnins Sergejs Kozlovics Lelde Lace Renars Liepins
Edgars Rencis Arturs Sprogis Andris Zarins
Institute of Mathematics and Computer Science, University of Latvia
Raina blvd. 29, LV-1459, Riga, Latvia
{janis.barzdins, karlis.cerans, mikus.grasmanis, audris.kalnins, sergejs.kozlovics, lelde.lace, renars.liepins, edgars.rencis, arturs.sprogis, andris.zarins}@lumii.lv

ABSTRACT

Nowadays, more and more issues need to be considered when implementing tools for domain specific languages with an orientation to the business process management. It is not enough to build just an editor for the language, various specific services need to be provided as well. In this paper, we describe our approach how to develop new domain specific languages for the mentioned field and their support tools. A description of, so called, transformation-driven architecture is outlined as well. It is shown how to use principles of the architecture in developing tool building platforms. Two domain specific languages together with tools implementing them are described as an example.

Keywords

Transformation-Driven Architecture, tool building platform, metamodels, model transformations, business process management.

1. INTRODUCTION

When talking about business processes and their role in development of information systems, an abbreviation BPM usually comes up. However, the meaning of BPM is not always the same. Initially, the letter M stood for Modeling, so with BPM everyone was to understand the development of tools being able to design business processes graphically. Later, the modelers' community realized it is not enough, and the second meaning of BPM arose – Business Process Management [1]. Now, business processes are not only modeled but also managed (meaning the process modeling tool had been integrated into some process management system which controls the process execution and integrates other parts of the information system).

Consequently, two kinds of graphical languages regarding business processes exist nowadays. Firstly, there are plenty of business process modeling languages. One of the most popular of them is probably the UML [2] activity diagrams. And secondly, there are also some business process management languages for which a compiler to some code executable on a process engine exists. Here, one must mention the BPMN (Business Process Modeling Notation [3]) and its possible target language – the BPEL (Business Process Execution Language [4]). A very important component of a BPMN tool is a compiler to the BPEL code being executable by some BPEL engine. According to the SOA ideology [1], a web service wrapper is developed for the information system components allowing the BPEL engine to manage execution of the whole system being allowed to be distributed through multiple companies.

However, most of these BPM languages or tools are often not very useful in everyday situations. Being very complex they are of course very useful for large enterprises. However, smaller and more specialized systems usually need only a small part of those facilities provided by the universal languages and tools. As a result, the usage of them tends to be too complicated. Moreover, tools (called the BPM suites) providing efficient and reliable implementation of process management languages and offering a whole set of support facilities are basically very expensive. Certainly, there are also some less expensive suites (e.g., BizAgi [5]) and cheap software-as-service offers by other vendors (e.g., Intalio [6]), but they are based on the same complicated languages and approaches. Therefore, specialized languages for narrow business domains are required, and that is where the DSLs (Domain Specific Languages) come into play. Although universal languages make advances towards specific tool builders (e.g., BPMN offers a possibility to add new attributes for tasks), they can never give such wide spectrum of facilities as DSLs can. In addition, frequently there are already well accepted notations for manual design of processes in some business domains, and they can be adequately formalized by the DSL approach. Moreover, buying and adapting some universal language or tool for one's small and specific case can often overcharge the benefits of using it afterwards. On the other hand, the development of a DSL can give little benefit if its implementations cost much. So we do have a need for some simple and unified way of building domain specific languages and tools. In this paper, we present a method of developing and implementing domain specific languages. Also, a success story of implementing two concrete DSLs is described here.

The paper is organized as follows. In Section 2, some possible requirements for tools implementing domain specific languages have been discussed. Two example tools ordered by real customers are introduced as well. Since they are to be parts of some information systems, some concrete services were to be satisfied by the tool. In Sections 3 and 4, our solution is presented. Besides that, the most important aspects of our metacase tool's architecture are sketched here as well. The approach of the architecture is demonstrated on the mentioned example tools. Section 5 concludes the paper.

2. TYPICAL REQUIREMENTS FOR DSL TOOLS IN THE FIELD OF BPM

2.1 DSL tools in general

When developing a tool for a domain specific language (a DSL tool) ordered by a customer, various needs have to be satisfied usually. Generally speaking, a DSL tool consists of two parts – a domain specific language it implements, and services it offers. So, various tools differ one from another not only in the notation of a DSL, but also in extent how easy they can be integrated in the outer world. Nowadays, life does not end with an editor of some domain specific language, it just starts there. In general, various types of functionality must be provided when designing a domain specific language including (but not limited to) a compiler to some target environment, a simulation feature, a debugger etc. However, when designing a DSL in the field of business process management, some more specific features come to mind. For a BPM domain specific tool to be successful, it must be able, for instance, to establish a connection to some external data source, for instance, a relational database. A DSL editor is often supposed to be a part of some larger information system, so it must provide facilities of collaboration with other parts of the system, the database being one of them. Besides, the collaboration must be possible in both design and run time of the tool. A crucial feature is also the ability to convert a process definition in this DSL into specification for some process execution engine in the system.

Other important issue to be considered is the ability to generate some kind of reports from the model information. DSL editors are often used to ease the preparation of, e.g., HTML or Microsoft Word documents containing information about the domain. So the tool should provide a way of generating such documents from the user-drawn diagrams.

Considering these issues is a crucial factor when designing a new DSL tool building platform. Some of the key facilities can be designed easily and added to the platform. However, others can be added later when such a necessity occurs. So, trying to satisfy the needs of different customers can play a great role in the growth of the platform. Therefore, in the next sections, we offer a description of two domain specific tools and explain their implementation within our DSL tool building platform.

2.2 Example tools

In this section, two concrete domain specific languages together with the tools implementing them will be discussed. First of them – Project Assessment Diagrams (further – PAD) – is an editor for visualizing business processes regarding review and assessment of submitted projects. This editor is based on UML activity diagrams and thus contains means for modeling business processes. Yet, some new attributes and some new elements have been added in order to handle the specific needs. For example, elements for controlling execution duration have been designed (elements *SetTimer* and *CheckTimer* that can be attached to a flow). The PAD editor has to be a part of a bigger information system for document flow management (a simplified BPM suite), so services providing interconnection between the system and the editor were needed. For example, a PAD model needs to be imported in a database where the information system can, for instance, make a trace for each client's project and then project this trace back to the editor for the visualization. This requirement was in some way similar to the business process monitoring performed, e.g., in ARIS [7] where groups of reasonably selected instances can be monitored. They go even further – a process mining is introduced to automate the monitoring process. So again – the problem has been known for some time already, but here we are trying to solve it by the means of a DSL instead of a universal language. Also, we do not need such powerful features providing the whole mining process. Instead, a very simple solution for business process monitoring was requested here.

The other domain specific language (and tool) we have developed is an editor for business processes in the State Social Insurance Agency (further – SSIA). Since users' habits were to be taken into account, this language syntactically is closer to BPMN. Again, specific services needed to be satisfied by the tool, three of which are the most worth mentioning:

- Online collaboration with a relational database – the searching for information in a database was to be combined with the graphical tool. The use case of that was a possibility to browse for normative acts during the diagram design phase – the normative acts are stored in a database and need to be accessed from the tool.
- Users wanted to start using the tool as soon as possible – even before the language definition has been fully completed. That means we need to assure the preservation of user-made models while the language can still change slightly. So the DSL evolution over the time is an issue to be considered.
- The tool must be able to generate some kind of reports from the visual information, preferably – in the format of Microsoft Word. Moreover, some text formatting possibilities must be provided in the tool, e.g., by ensuring the rich text support to input fields.

Besides those, some more minor issues were highlighted during the DSL design phase, but we are not going to cover all of them here due to the space limitation.

It must be mentioned that, when designing languages, the main emphasis was put on the fact that processes must be easy perceived by the user. At the same time, however, languages had to be suitable for serving as process management languages without any changes. Since languages have been designed in such a manner, it is possible to integrate them into a full-scale BPM suite later. There the process definitions will be used to manage the document flows in a typical to BPM manner.

3. IMPLEMENTATION BACKGROUND

3.1 General ideas

We have used our metamodel-based Graphical Tool-building Platform GrTP [8] to implement the domain specific languages PAD and SSIA. The recent version of GrTP is based on principles of the Transformation-Driven Architecture (TDA, [9]). In this Section, the key principles of the TDA and GrTP as well as their applications in DSL implementation are discussed.

3.2 The Transformation-Driven Architecture

The Transformation-Driven Architecture is a metamodel-based approach for system (in particular, tool) building, where the system metamodel consists of one or more interface metamodels served by the corresponding engines (called, the interface engines) and the (optional) Domain Metamodel. There is also the Core Metamodel (fixed) with the corresponding Head Engine. Model transformations are used for linking instances of the mentioned metamodels (see Fig. 1).

The Head Engine is a special engine, whose role is to provide services for transformations as well as for interface engines. For instance, when a user event (such as a mouse click) occurs in some interface engine, the Head Engine may be asked to call the corresponding transformation for handling this event. Also, a transformation may give commands to interface engines. Thus, the Core Metamodel contains classes Event and Command, and the Head Engine is used as an event/command manager.

Since it has been published in [9], we won't go into details about TDA here. Instead, we will just outline the main technical assumptions for TDA in order to set the background:

- The model data are stored in some repository (like EMF [10], JGraLab [11] or Sesame [12]) with fixed API (Application Programming Interface).
- The API of the repository should be available for one or more high-level programming languages (such as C++ or Java), in which interface engines will be written.
- Model transformations may be written in any language (for instance, any textual language from the Lx family [13] or the graphical language MOLA [14] may be used). However, the transformation compiler/interpreter should use the same repository API as the engines.
- When a transformation is called, its behavior depends only on the data stored in the repository.
- Only one module (transformation or engine) is allowed to access the repository at the same time. Concurrency and locking issues are not considered.

We have developed a, so called, TDA framework which implements the principles of the TDA. The TDA framework contains one predefined engine – the head engine – and the repository (we are using our very efficient in-memory repository [15] with a fixed API being available from the programming language C++ in which engines are to be written). Other interface engines may also be written and plugged-in, when needed. The TDA framework is common to all the tool building platforms built upon the TDA. The framework is brought to life by means of model transformations. One can choose between writing different transformations for different tools and writing one configurable transformation covering several tools.

Actually, one more layer is introduced between the model transformations and the repository. It is called the repository proxy and it contains several features being common for all tool building platforms built upon the TDA. The most notable of them is perhaps the UNDO/REDO functionality – since it is embedded in the proxy, engines and transformations do not have to consider the UNDO and REDO actions. All the commands are intercepted by the proxy and then passed further to the repository.

3.3 The TDA-based Tool Building Platform GrTP

Next, we have developed a concrete tool building platform called the GrTP by taking the TDA framework and filling it with several interfaces. Besides the core interface, five more interfaces have been developed and plugged into the platform in the case of GrTP:

- The graph diagram interface is perhaps the main interface from the end user's point of view. It allows user to view models visually in a form of graph diagrams. The graph diagram engine [16] embodies advanced graph drawing and layouting algorithms ([17, 18]) as well as effective internal diagram representation structures allowing one to handle the

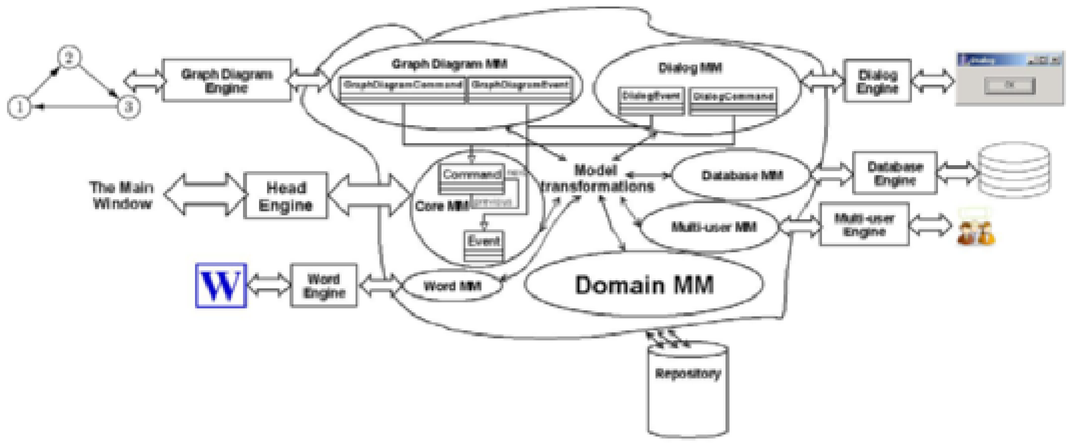


Figure 1. The Transformation-Driven Architecture framework filled with some interfaces.

visualization tasks efficiently even for large diagrams.

- The property dialog interface allows user to communicate with the repository using visual dialog windows.
- The database interface ensures a communication between the model repository and a database.
- The multi-user interface performs the task of turning a project into a multi-user project and considers other issues regarding that.
- The Word interface helps user to establish a connection to Microsoft Word and to send data to it.

The final step is to develop a concrete tool within the GrTP. This is being done by providing model transformations responding to user-created events. A fair part of these transformations usually tend to be universal enough to be taken from our already existing transformation library instead of writing them from scratch (transformations responding to main events like creating new element, reconnecting line ends, making a mouse click or double click etc., as well as such platform specific transformations as copy, cut, paste, delete, import, export, etc.). In order to reduce the work of writing transformations needed for some concrete tool, we introduce a tool definition metamodel (TDMM) with a corresponding extension mechanism. We use a universal transformation to interpret the TDMM and its extension thus obtaining concrete tools working in such an interpreting mode. This is explained a bit more in the next subsection.

3.4 The tool definition metamodel and its usage for building concrete tools

First of all, we explain the way of coding models in domain specific languages. The main idea is depicted in Fig. 2. As can be seen here, the graph diagram metamodel (conforming the one from Fig. 1) is complemented with types turning a general graph diagram into a diagram of some concrete tool (e.g., some business process editor). A model here is a set of graph diagrams every one of which consists of elements – nodes and edges. An element in its turn can contain several compartments. At runtime, each visual element (diagrams, nodes, edges, compartments) is attached to exactly one type instance (see classes *DiagramType*, *ElementType*, *CompartmentType*) and to exactly one style instance. Here, types can be perceived as an abstract syntax of the model while the concrete syntax being coded through styles.

Now, about the proposed tool definition metamodel. The main idea of the tool definition metamodel together with the extension mechanism is presented in Fig. 3. Apart from types, the tool definition metamodel contains several extra classes describing the tool context (e.g., classes like *Palette*, *PopUp*, *ToolBar*, etc.). Moreover, the tool definition metamodel contains, so called, extension mechanism providing a possibility to change behavior of tools represented by the metamodel. The extension mechanism is a set of precisely defined extension points through which one can specify transformations to be called in various cases. One example of a possible extension could be an "AfterElementCreated" extension providing the transformation to be called when some new element has been created in a graph diagram. Tools are being represented by instances of the TDMM by interpreting them at runtime. Therefore, to build a concrete tool actually means to generate an appropriate instance of the TDMM and to write model transformations for extension points. In such a way, the standard part of any tool is included in the tool definition metamodel meaning that no transformation needs to be written for that part. Instead, an instance of the TDMM needs to be generated using a graphical configurator. At the same time, the connection with the outer world (e.g., a database or a text processor) is established by writing specific model transformations and using the extension mechanism to integrate them into the TDMM.

3.5 Benefits of the TDA

The main advantage of the transformation-driven architecture is its idea of providing explicit metamodeling foundations in building tools for domain specific languages. Although there

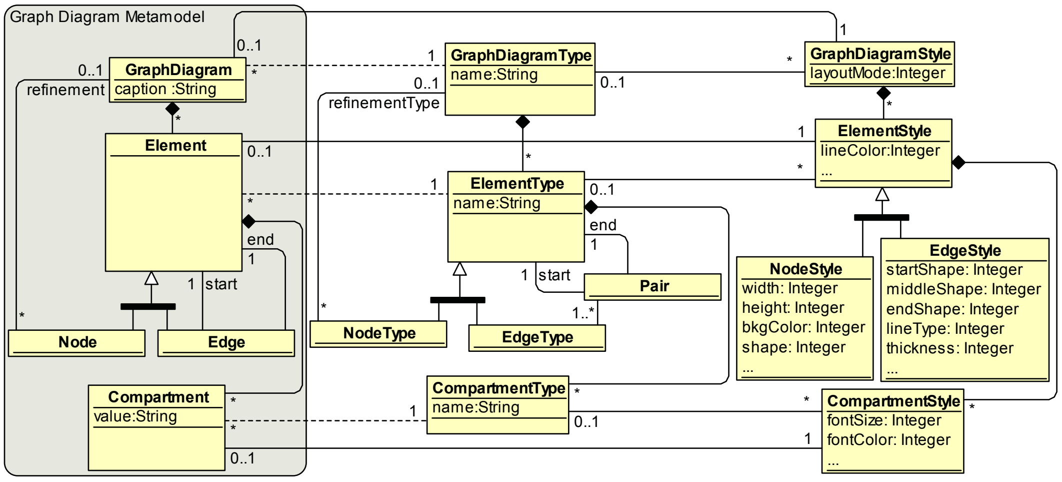


Figure 2. The way of coding models.

already exist some metacase tools accepting the idea of DSL tool definition by a metamodel (e.g., Eclipse GMF [19] and Microsoft DSL Tools [20]), they generally offer only some configuration facilities allowing definition of a DSL tool in a user-friendly way while a direct access to metamodels is either limited or provided using some low level facilities. The pace-maker of the field is perhaps the Metacase company whose product MetaEdit+ [21] provides Graph, Object, Property, Port, Relationship and Role tools to ensure the easy configuration of concrete domain specific tools. Another well-known example is Pounamu/Marama [22, 23] which offers shape designer, metamodel designer, event handler designer and view designer to obtain a DSL tool. On the contrary, the TDA is completely transparent meaning a user can have a free read and write access to its metamodels and their instances. Of course, extra services like graphical configurator of a DSL tool can be offered as well, but the user is not forced to use it. It must be underlined that, if following the TDA, the definition of a concrete domain specific tool only involves developing model transformations and nothing else. The TDA follows the ideas of the MDA [24] stating that the common part of syntax and semantics can be formalized through a metamodel. The whole specific part at the same time can be put into model transformations.

The other notable advantage of the TDA is its ability to get in touch with the outer world. This is being done by adding new engines to the TDA framework. Since there is no need to go deep in implementation details of other engines or other parts of the TDA, this is considered to be a comparatively easy task.

4. The development of PAD and SSIA using GrTP

Besides the trivial part – generation of a tool definition metamodel's instance forming the graphical core of the tool – we decided to develop three more engines we did not have at that moment. Those engines were the database engine, the multi-user engine and the Word engine. Since the TDA framework provides a possibility to plug in new interfaces (engines together with their respective metamodels) easily, the development and integration of the engines was done quite harmlessly. We must admit there were some difficulties to integrate the multi-user interface, however they were mostly of technical nature – the tool definition metamodel had to be changed a bit as well.

Next, according to the extension mechanism, some specific transformations needed to be written in order to put a life into the static tools – to make them dynamic. These transformations referred to generating, for instance, the correct items for combo boxes, to changing items in context menus dynamically, to assigning the correct styles to visual elements (although this can be partly specified in the static part as well) etc. These transformations had to be written and attached to appropriate

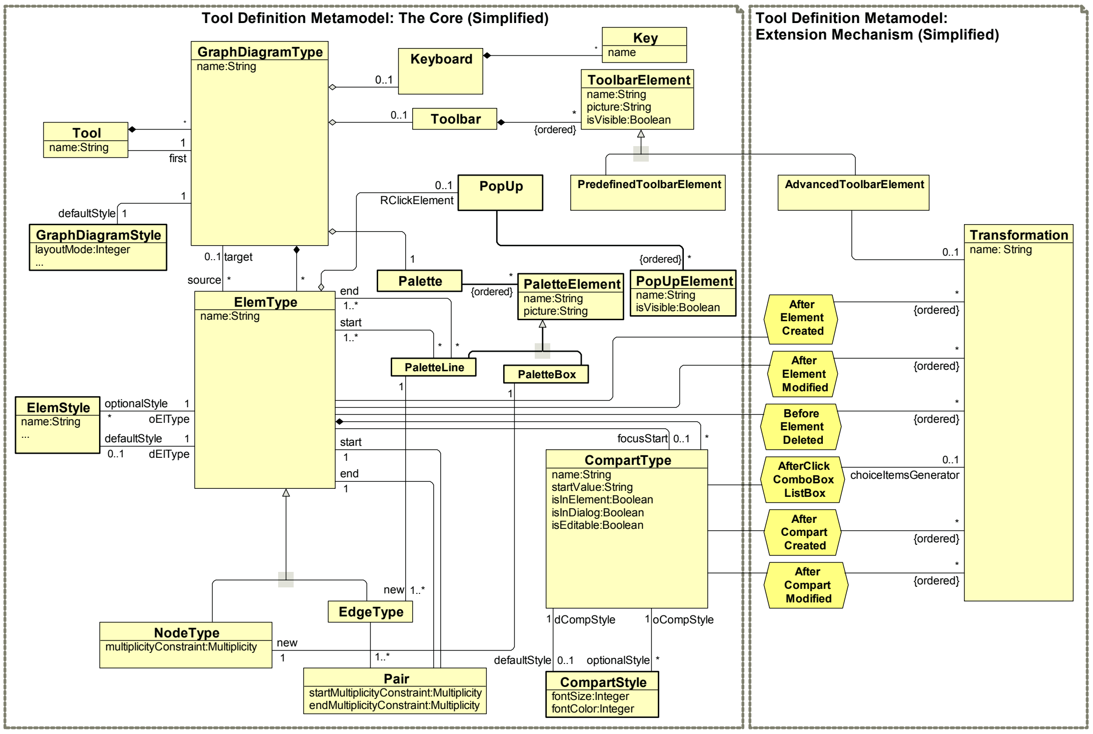


Figure 3. Basic principles of the tool definition metamodel.

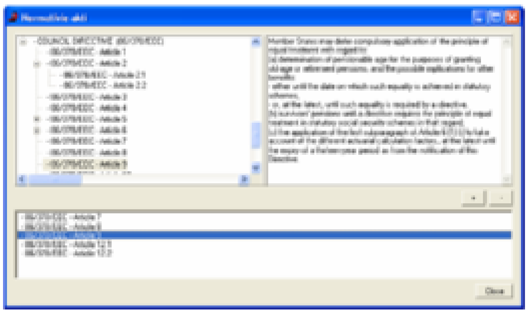


Figure 4. Browsing for normative acts stored in the database from the tool's dialog window.

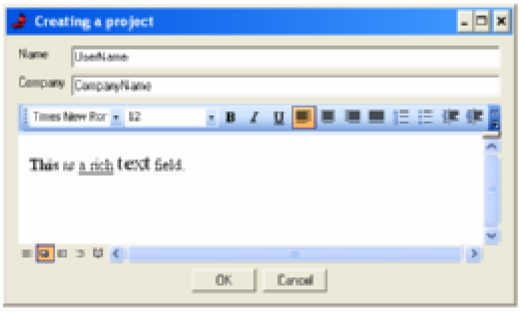
extension points thus forming the concrete tool.

The most challenging part of the development of tools was perhaps the ensuring interconnection between the tool and a relational database. Since the graphical tool was meant to be just one piece of the whole information system's software, it was already clear before that this problem will have to be faced sooner or later. The issue has been classic in the world of workflows – some business process has been being modeled in a tool and passed to a relational database afterwards. The information system would then take care of applying the process to individual clients and storing the history of how far each client has gone through the process. On request, the tool should be able to visualize the history for some particular client as well.

The classic solution of the problem advises using the ORM method (Object-relational mapping, [25]) stating that a simple mapping between the model repository and the database must be made and a generation of metamodel instances and/or database records must be performed. However, this solution was not acceptable in our case because of its limitations in the process of generation of instances as a response to a query for the database – the types or return tables must be known before. At the same time, the SSIA project required the possibility of receiving answer to an arbitrary query.

Our solution included turning a part of the metamodel storing the business processes together with their respective element types and styles (see Fig. 2) into a database schema. That was a pretty straightforward job – if abstracting from the details, the database was made to store data in RDF format [26]. So, all the database engine had to do was generating the contents of the database from the model and vice versa. Next, a translator was made in an information system part of the system carrying out a connection between the RDF-type database and the actual database of the system. Thus, by introducing such an intermediate layer between the tool and the actual database, the database engine was to be written once and for all – it does not depend on the actual database schema.

Eventually, the tools obtained in GrTP satisfied all the needs customers had highlighted, including the ones mentioned in Section 2. The connection to the relational database provided by the database engine ensured fast information searching capabilities in database in combination with the tool. Thus, an easy browsing for information stored in the relational database (in



Creating a project

Name UserName

Company CompanyName

This is a rich text field.

Figure 5. A rich text component.

this case – normative acts) was possible from the tool interface (see Fig. 4). Next, an add-only DSL evolution comes at no extra cost if using the tool definition metamodel to develop tools in GrTP. Indeed, if the DSL demands some more element or compartment types to be added, we just add new instances to *ElementType* or *CompartmentType* classes in the model coding metamodel. Since it has nothing to do with already existing types of the language, existing models remain unmodified. This can be achieved because of the fact that we store the DSL definition in the same modeling level with the actual models – the connection between a model element and its definition is obtained without crossing levels. However, if changes in DSL are not of add-only type, some extra work needs to be done – elements and compartments of old types may need to be either deleted of relinked to some new types (see dashed associations in Fig. 2). In our framework, all this work can be done by model transformations. It must be mentioned that add-only changes are comparatively easy to implement in most metacase tools, although not in all. For example, it is still a quite tough problem in tools based on JGraLab repository [11].

Finally, a report generator was built using the Word engine. It introduces a simple graphical language allowing one to specify the information to be put in a Microsoft Word document. In the engine, several extra services were implemented as well. For example, a Word window was embedded in a property dialog windows generated by the property dialog engine and providing a possibility for a user to create rich text compartment values as was requested by the customer (see Fig. 5).

5. CONCLUSIONS

In this paper, we described our approach how to develop new domain specific languages and tools for supporting them. A short description of the transformation-driven architecture and its framework was outlined as well. The architecture was illustrated in its application in the graphical tool building platform GrTP upon which two concrete domain specific languages were implemented.

It was mentioned that nowadays DSL is often to be only a part of some bigger information system and the tool supporting it thus must be able to communicate with the outer world. In fact, this approach is only one of the possible solutions for the problem of how to integrate the tool with other parts of an information system. The other possible way is to develop so called business process management suites in which all the necessary features are included. It involves also the issue of how to include fragments of existing information systems into a tool environment. It is usually done by turning components of the information system into a web service thus providing an appropriate network addressable application program interface for it.

Considering the issues mentioned above, our closest goal is to develop a platform for building domain specific suites. One of the possible domains for the approach could be suites incorporating process and document management integrated with sophisticated document generation procedures. Though large-scale expensive solutions such as EMC Documentum exist here, a very appropriate niche for small-scale, but logically sophisticated DSL-based solutions could be document management in various government institutions. The web service based approach will ensure very tight integration of the DSL execution environment with the rest of the suite including full access to databases.

6. REFERENCES

[1] J. F. Chang. Business Process Management Systems. Auerbach Publications, 2006, pp. 286.

[2] UML, http://www.uml.org.

[3] BPMN, http://www.bpmn.org.

[4] BPEL, http://www.bpelsource.com.

[5] BizAgi, http://www.bizagi.com.

[6] Intalio, http://www.intalio.com.

[7] ARIS platform, http://www.ids-scheer.com/en/ARIS_ARIS_Software/3730.html.

[8] J. Barzdins, A. Zarins, K. Cerans, A. Kalnins, E. Rencis, L. Lace, R. Liepins, A. Sprogis. GrTP: Transformation Based Graphical Tool Building Platform. *Proc. of Workshop on Model Driven Development of Advanced User Interfaces, MODELS 2007*, Nashville, USA.

[9] J. Barzdins, S. Kozlovics, E. Rencis. The Transformation-Driven Architecture. *Proceedings of DSM'08 Workshop of OOPSLA 2008*, Nashville, USA, 2008, pp. 60–63.

[10] Eclipse Modeling Framework (EMF, Eclipse Modeling subproject), http://www.eclipse.org/emf.

[11] S. Kahle. JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek f¨ur TGraphen, *Diplomarbeit*, University of Koblenz-Landau, Institute for Software Technology, 2006.

[12] Sesame, http://www.openrdf.org, 2007.

[13] J. Barzdins, A. Kalnins, E. Rencis, S. Rikacovs. Model Transformation Languages and their Implementation by Bootstrapping Method. *Pillars of Computer Science*, LNCS, vol. 4800, Springer-Verlag, 2008, pp. 130-145.

[14] A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA, *Proceedings of MDAFA 2004*, LNCS, vol. 3599, Springer-Verlag, 2005, pp. 62-76.

[15] J. Barzdins, G. Barzdins, R. Balodis, K. Cerans, A. Kalnins, M. Opmanis, K. Podnieks. Towards Semantic Latvia. *Communications of the 7th International Baltic Conference on Databases and Information Systems (Baltic DB&IS'2006)*, Vilnius, 2006, pp. 203-218.

[16] J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, A. Zarins. A Graph Diagram Engine for the Transformation-Driven Architecture. *Proceedings of MDDAUI'09 Workshop of International Conference on Intelligent User Interfaces 2009*, Sanibel Island, Florida, USA, 2009, pp. 29-32.

[17] P. Kikusts, P. Rucevskis. Layout Algorithms of Graph-Like Diagrams for GRADE Windows Graphic Editors. *Proceedings of Graph Drawing '95*, LNCS, vol. 1027, Springer-Verlag, 1996, pp. 361–364.

[18] K. Freivalds, P. Kikusts. Optimum Layout Adjustment Supporting Ordering Constraints in Graph-Like Diagram Drawing. *Proceedings of The Latvian Academy of Sciences*, Section B, vol. 55, No. 1, 2001, pp. 43–51.

[19] A. Shatalin, A. Tikhomirov. Graphical Modeling Framework Architecture Overview. *Eclipse Modeling* Symposium, 2006.

[20] S. Cook, G. Jones, S. Kent, A. C. Wills. Domain-Specific Development with Visual Studio DSL Tools, Addison-Wesley, 2007.

[21] MetaEdit+, http://www.metacase.com.

[22] N. Zhu1, J. Grundy, J. Hosking. Pounamu: a meta-tool for multiview visual language environment construction. *Proc. IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC'04)*, 2004, pp. 254-256.

[23] J. Grundy, J. Hosking, N. Zhu1, N. Liu. Generating Domain-Specific Visual Language Editors from High-level Tool Specifications. *21st IEEE International Conference on Automated Software Engineering (ASE'06)*, 2006, pp. 25-36.

[24] MDA Guide Version 1.0.1. OMG, http://www.omg.org/docs/omg/03-06-01.pdf.

[25] C. Richardson. POJOs In Action. Manning Publications Co, 2006, pp. 560.

[26] Resource Definition Framework, http://www.w3.org/RDF.

Use of a Domain Specific Modeling Language for Realizing Versatile Dashboards

Ulrich Frank
ulrich.frank@uni-due.de

David Heise
david.heise@uni-due.de

Heiko Kattenstroth
heiko.kattenstroth@uni-due.de

Chair of Information Systems and Enterprise Modeling
University of Duisburg-Essen
Universitaetsstr. 9, 45141 Essen, Germany

ABSTRACT

In order to make performance indicators a useful instrument to support managerial decision making, there is need to thoroughly analyse the business context indicators are used in as well as their mutual dependencies. For this purpose, it is recommended to design indicator systems that do not only include dedicated specifications of indicators, but that account for relevant relationships. In this paper, a DSML is proposed that enables the convenient design of consistent indicator systems at type level, which supports various kinds of analyses, and can serve as conceptual foundation for corresponding performance management systems, such as dashboard systems. Furthermore, indicator systems may also be used during run-time at the instance level to promote the distinguished interpretation of particular indicator values.

Keywords

Domain-Specific Modeling Language, Enterprise Modeling, Performance Management, KPI

1. MOTIVATION

In recent years, the increasing appreciation for performance indicators has promoted the idea of systems that provide users with performance related data. These systems, which we refer to as 'Performance Management Information Systems' (PMIS), are supposed to inform the individual user at a quick glance about the performance of entities such as an entire firm, specific business units, business processes, resources, and IT services. Inspired by technical metaphors such as 'cockpit' or 'dashboard', PMIS are more and more considered as a general instrument to foster managerial action, especially with respect to supporting, measuring, and monitoring decisions. The design of a PMIS implies the conception of indicators and systems of interrelated indicators ('indicator systems'). Indicator systems are usually defined by (top) management – with no regard of how they could be represented in an information system.

Current PMIS, such as dashboards, predominantly focus on the visualization of indicators that are considered to be relevant for certain decision scenarios. For this purpose, dashboard systems provide generic visualization 'gadgets', e.g., speedometers, traffic lights, or bar charts, that are usually applied to data originating from databases or files. However, to design PMIS that effectively support mangerial decision making, focusing on visualization only is not sufficient [3]. Instead, there is need to analyze what concepts are required to structure and effectively support a targeted decision.

Moreover, the design of indicator systems is not trivial. Already the specification of an indicator does not only require a profound understanding of the corresponding decision scenario and the relations to other indicators, but also recommends taking into account how an indicator affects managerial decision making [16, 19]; if managers regard an indicator as an end in itself, it will result in opportunistic actions that are likely not compliant with the objectives of a firm. This is even more important, because managers and other stakeholders are incited to predominantly align their behavior with specific (maybe mandatory) indicators and associated target values only [12, 17, 20]. If PMIS do not adequately address these challenges, they are likely to fail their purpose.

In this paper, we present an approach for PMIS that incorporates a domain-specific modeling language (DSML) for designing expressive and comprehensible indicator systems as core element. The DSML, called ScoreML, aims at promoting transparency, especially with regard to counter dysfunctional effects of indicators such as opportunistic behaviour. Also, indicator systems created with the ScoreML serve as a conceptual foundation for developing corresponding software. In addition to this use at build-time, our approach makes use of indicator systems at run-time as well, for example, as a front end to instance level performance data.

The approach is based on a comprehensive method for enterprise modeling and consists of the following components:

- a domain-specific modeling method comprising a language for modeling indicator systems – the ScoreML – and a corresponding process model that guides its application;
- a modeling environment implementing a ScoreML editor that is integrated with further editors for domain-specific modeling languages that are part of the enterprise modeling method;
- a software architecture for PMIS, in which the modeling environment constitutes the core component and that allows for integration with existing information systems.

Figure 1 illustrates the components of the PMIS. In this paper, we focus on the modeling language and its utilization in the context of the envisioned systems architecture. The other components are briefly discussed. The remainder is structured as follows: We derive domain-specific requirements for PMIS in Section 2. The prospects of our approach are illustrated in Section 3. The conceptual foundation, i.e.,

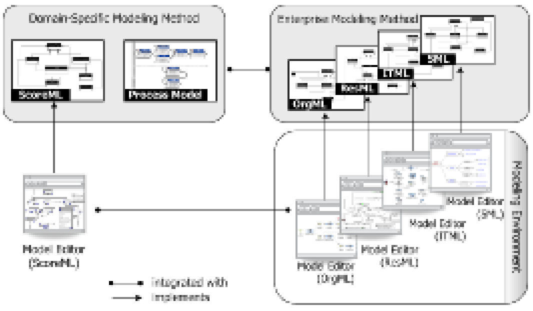


Figure 1: Components of the PMIS

meta-model and language architecture of the ScoreML, are presented in Section 4; the architecture for a model-based PMIS is envisioned in Section 5. Related work is discussed in Section 6. The paper closes with an evaluation, concluding remarks, and an outlook on future work in Section 7.

2. PMIS: REQUIREMENTS

An analysis of the current practice of dealing with indicators reveals a number of shortcomings. Based on these deficiencies, requirements for the domain-specific modeling language as well as for the envisioned architecture of a PMIS, which implements the DSML and integrates it with existing tools, can be derived.

First, currently there is hardly support for systematically creating and maintaining indicator systems available – indicators or indicator systems that are suggested in pertinent literature are usually described by informal concepts (e.g., [11, 15]). Hence, there is no linguistic support for guiding the construction of coherent indicator systems. This is a severe shortcoming: If an indicator system is partially inconsistent – e.g., includes incomplete indicator descriptions, undocumented dependencies, or even contradicting indicators – it jeopardizes its very purpose.

Req. 1 – Design of Indicator Systems: The design of consistent indicator systems should be promoted – if not enforced.

Second, the interpretation of indicators is crucial for well-founded decision-making. An adequate use of an indicator system implies a knowledgeable interpretation of the numbers that represent an indicator. Otherwise, a focus on 'meeting the numbers' (cf. [17]) may result in opportunistic behaviour, promote misleading conclusions and unfortunate decisions that – in the worst case – impede the overall enterprise performance.

Req. 2 – Business Context: To support the user with an appropriate interpretation of indicators, the indicator systems should be enriched with relevant context information to enhance the interpretation of indicators. This requires not only offering concepts that represent indicators, but also allowing for associating them with concepts that represent the business context (such as business processes).

Third, the utilization of indicator systems affects an enterprise and its employees at various – if not all – organizational levels, e.g., from executives at the strategic level to business units or IT experts at the operational level. The specific perspectives and levels of expertise vary among these groups of stakeholders. For instance, a process manager will have expectations that are different from those of an IT manager or an executive with respect to the types of indicators as well as the levels of detail and abstraction.

Req. 3 – Stakeholders: Meaningful presentations at different levels of abstraction are required to satisfy the needs of the multiple groups of prospective users. To foster an intuitive use of the language, concepts should be provided that these groups are familiar with.

Fourth, indicators used at different organizational levels are usually interrelated in that an indicator at a higher organizational level (e.g., strategic level) is often calculated from indicators at lower organizational levels (e.g., operational level). If not interrelated directly, indicator types can still be related indirectly, especially if the objects they meaasure are interrelated. Indicators, for instance, that measure the performance of business processes might be dependent on indicators measuring the performance of an information system underlying these processes, and thus are indirectly interrelated.

Req. 4 – Cross-Disciplinary Analyses: It should be possible to analyze interdependencies between indicators associated with different perspectives. This allows for making decisions on a more profound information base and for considering dependencies that go beyond the indicator system itself. Note that this request corresponds to the idea of the Balanced ScoreCard [12].

Fifth, supporting decisions requires particular indicator values, i.e., instance level data. There is a wide range of tools that aim at preparing and presenting these values, e.g., from dataware house to monitoring to reporting tools. However, they usually do not support users in interpretation and assessment of the presented values. Associating indicator values with the corresponding conceptual level – i.e., with the indicator system they are instantiated from and that are integrated with the relevant business context (cf. *Req. 2*) – contributes to a more sophisticated appreciation of indicator values.

Req. 5 – Instance Data: Tools for modeling indicator systems should be integrated with systems that manage corresponding instance level data (or integrate a corresponding component). It should be possible to navigate from the instance level to the conceptual level – and vice versa.

Sixth, PMIS usually visualize indicators in various ways. However, the cognitive styles of the involved users differ. Furthermore, different decision scenarios require different visualizations [2, 4]. In some cases, already the fact that an indicator is over (or below) a pre-defined threshold matters. In other cases, the focus is on performance over time, or the measured indicator needs to be compared to pre-defined thresholds.

Req. no.	*Description of Requirement*
Req. 1	Promote design of consistent indicator systems
Req. 2	Offer concepts for business context
Req. 3	Provide abstractions for different stakeholders
Req. 4	Enable cross-disciplinary analyses
Req. 5	Integrate type and instance level
Req. 6	Enable user-specific visualizations

Table 1: Summary of Requirements

Req. 6 – Visualization: Different stakeholders and different decision scenarios demand for versatile graphical representations of indicators. Therefore, it should be possible to adapt graphical visualisations to the individual needs of stakeholders without compromising the semantics of the represented concepts.

Table 1 summarizes the requirements for a performance management information system.

3. PROSPECTS OF THE APPROACH

The requirements pose the demand for an approach that supports the design and utilization of indicator systems in a systematic and structured manner. Conceptual models seem to be suitable, since they promise to reduce complexity by focusing on those aspects that are essential – and abstract from other. In this regard, the ScoreML promises more consistent indicator systems and allows for various analyses that – without such a support – a user can hardly perform. In the following, we illustrate the envisioned use of the DSML for designing and utilizing indicator systems at build-time as well as its potential for being leveraged as a 'dashboard' during run-time.

3.1 Focus on Build-Time

Users design indicator systems with the ScoreML by choosing indicators they consider relevant and adquate to support the targeted decision. At first, these indicators will be described on a more abstract level that is usually hardly quantifiable, e.g. "competitiveness". They can then refine these high-level indicators until they get down to a set of indicators that allow for expressive quantifications. By associating them to the objects they refer to, i.e., the reference objects they measure, the indicators are enriched with additional context information (cf. *Req. 2*). Once the indicator system is designed, (yet undiscovered) interdependencies among indicators can be elicited.

Figure 2 exemplifies this procedure for an IT Manager. It shows an indicator system model (top) and an excerpt of integrated IT resource/business process models (bottom). In the indicator system model, a few indicator types (attributes are omitted) for an IT-related indicator system are displayed. The indicator type *efficiency of IT department* is calculated from *IT costs* and *IT operations efficiency* (calculation rule is omitted, too). Some indicator types are associated to reference objects (an IT resource and two business process types). Here, different perspectives on an enterprise are accounted for: Two indicator types that are not directly interrelated (*IT costs* from an IT perspective and *average throughput time* from an operations perspective's indicator

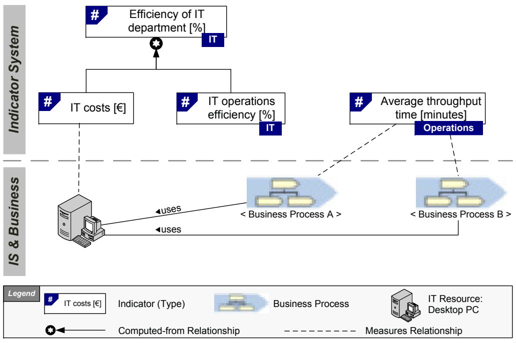


Figure 2: Short example including notation

system) are indirectly related; hence, an IT manager focusing on improving (e.g., reducing) *IT costs* only might, in the end, impede the performance of a related business process type's *throughput time*. If a timely execution of the process is more important than the IT costs of this resource, the additional business context information (cf. *Ref. 2*) and the capability to navigate through the other models (e.g., business process or IT resource models; cf.*Req. 4*) fosters decision-making (in this case of the IT manager). With regard to the ScoreML, the IT Manager could establish a specific association type between *IT costs* and *average throughput time* (e.g., 'influences') that visualizes the potential cause and effect relationship between these indicator types for further utilizations of the indicator system.[1]

3.2 Focus on Run-Time

Besides using models of indicator systems and the related models of the business context at build-time, they can also be used at run-time. A simplified example of an application scenario is illustrated in Figure 3.

A process owner, who is responsible for an online sales process, uses his personal dashboard to monitor the performance of the process. While the *daily revenue* corresponds to his expectations, the *average throughput time* (time between ordering and notification of the customer that the order has been approved) is exceeding its threshold. As a consequence, the currently running (active) instances of this process are affected, which are depicted in the lower section of Figure 3a.

To get a better understanding of the reasons for the dissatisfactory performance, he investigates ('drill-down') the indicators on which the critical indicator *average throughput time* depends, i.e., is calculated from. An example model of a business process *Online Sales* along with the required IT resources is displayed in Figure 3b.

The *process payment* activity is not functioning properly because the required part of the ERP-system is not available. The process owner escalates the problem to the IT staff, e.g.,

[1]Further, more extensive examples of indicator system models by means of ScoreML can be found in [7] and at http://openmodels.org/node/190.

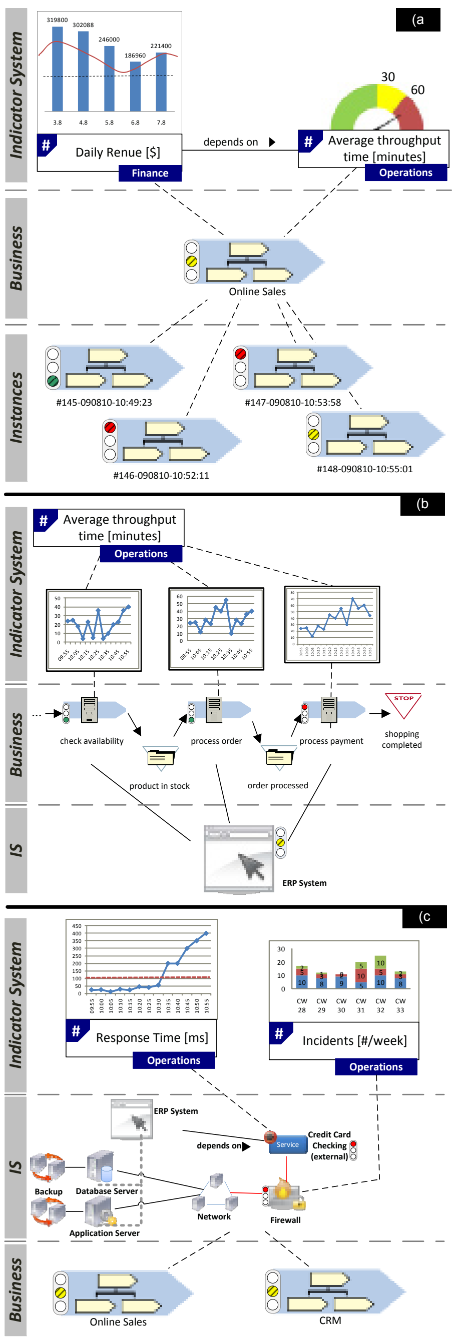


Figure 3: Dashboard for an Online Sales Process

through a ticket that refers to the particular business process and the malfunctioning IS. A member of the IT staff receives a notification about the incident. Hence, he uses his dashboard to assess the business impact (e.g., how many business processes are impacted? What is the loss of revenues to be expected in case of an outage?) of the incident. An excerpt of a model of the ERP system is displayed in Figure 3c along with corresponding indicators. The service *credit card checking* is offered by an external partner and securely accessed through a firewall. Obviously, the connection is not stable, i.e., has a high response time. Furthermore, the firewall was subject of several severe problems ('incidents') in the past weeks. Based on the information available, the user can decide what to do next, e.g., contact the vendor of the firewall and demand for a satisfactory solution.

4. MODEL-BASED PMIS

The ScoreML is based on a formal syntax and precise semantics, which provides two advantages over non-formal or general purpose approaches: First, it effectively supports and constrains users to build consistent and syntactically correct models (cf. *Req. 1*) as well as it facilitates convenient, intuitive, and secure modeling. Second, a DSML enables various kinds of analyses and transformations including code generation for corresponding software (cf.[13]). Furthermore, the ScoreML comprises a graphical notation with specialized, interchangable icons, which fosters communication between stakeholders with different professional backgrounds (cf.*Req. 3*).

4.1 Language Architecture

The approach we chose to develop the DSML is to enhance an existing method for enterprise modeling (EM) – the *multi-perspective enterprise modeling* (MEMO)-method [5] – by concepts and further components for designing and utilising indicator systems. MEMO consists of an extensible set of domain-specific modeling languages meant to model different aspects of an enterprise, such as corporate stratetgy (with the Strategy Modeling Language, SML; [8]), business processes (Organization Modeling Language, OrgML; [5]), resources (Resource Modeling Language, ResML; [10]), or IT resources (IT Modeling Language, ITML; [14]). MEMO is multi-perspective since it provides different views on various aspects of an enterprise. The MEMO languages are integrated in two ways (cf. [6]): First, they are integrated by a common meta meta model (M_3), so that the DSMLs are based on the same language specification (meta language). Second, they share common concepts at the meta level (M_2), which enables the integration of different perspectives (and models) addressed by each DSML (e.g., a meta concept 'business process' in OrgML and ITML).

Figure 4 illustrates the language architecture of MEMO and the interrelations between the DSMLs and the corresponding models at type level (M_1). Integrating the proposed DSML with the MEMO framework allows to benefit from a variety of existing modeling languages. Thereby, it is possible to associate indicator systems with models of the business context (cf.*Req. 2*) and representations different groups of stakeholders are familiar with (cf. *Req. 3*). Furthermore, the integration of the models provides a foundation to enable cross-disciplinary analyses that span various perspectives (cf. *Req. 4*).

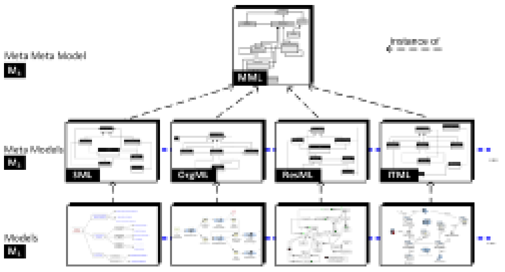


Figure 4: MEMO Language Architecture

4.2 Language Specification: ScoreML

The DSML is specified in a meta model using the Meta Modeling Language MML (cf. [6]). Figure 5 shows an excerpt of the ScoreML's meta model and depicts its main concepts. The specification of the language faced a number of challenges. Three important issues are addressed below.

First, ScoreML has to provide the users with a precise conception of indicators. From a modeling perspective, an indicator can be an attribute of an object, e.g., 'IT costs' of a piece of hardware; an aggregation of attributes of a collection of objects, e.g., the sum of 'IT costs' of all IT resources; or it can represent a snapshot of certain states over time, e.g., 'Average monthly IT costs'. In order to provide the user with convenient modeling concepts, we decided to introduce a specific abstraction rather than regarding indicators as attributes of objects or object collections. Such a conception includes core attributes and a differentiated conception of relationships between indicator types. This is realized by the meta type *Indicator* that comprises predefined attributes (e.g., name and description, purpose, potential bias) and a set of self-reflexive association types (e.g., computed from or similar to). We further introduced the association type *CustomizedRelationship* that enables users to qualify additional relations between indicators – for instance, an indicator can have an effect on another indicator (cf. Section 3.1). Additional customized attributes and non-'1..1'-associations can be realized by the meta types 'IndicAttributes' and 'IndicLink'.[2]

The second decision pertains to the flexibility and adaptability of indicators. The ScoreML has to allow users for adapting indicators to their individual needs and, furthermore, enrich indicators with additional semantics concerning the context they are used in. The former is addressed by distinguishing the concept indicator into the meta types *Indicator* and *SpecificIndicator*: While instances of *Indicator* represent generic information about an indicator type, *SpecificIndicator* allows users to assign this indicator type to specific reference object types, e.g., an indicator type 'average throughput time' assigned to 'business process type A' and to 'business process type B', including different values in the attributes benchmark, (avg.) value etc. In this context, the meta type *Threshold* allows for defining user-specific thresholds and corresponding notifications for a *SpecificIndicator*. This enables users to develop their individual 'performance dashboard' that includes indicators, thresholds, corresponding notifications and visualizations, and that fits to their personal cognitive style (cf. *Req. 6*). The latter – the need for additional semantics – is tackled by the meta types *ReferenceObject*, which is a surrogate for meta types like *BusinessProcess*, *Resource*, *Product* etc., and *DecisionScenario*, which enables the mapping of indicator types to scenario types (e.g., 'assessment of IT resources'). Note, the surrogate serves to illustrate the integration of ScoreML with the other MEMO languages: An indicator can be assigned to each (reasonable) meta type in one of the other DSMLs, which is the foundation for performing cross-disciplinary analyses. The re-use of concepts from other languages is denoted in the meta model with a rectangle at the concepts headers, including information about the origin (see the color legend in Figure 5).

Third, it is required to differentiate between types and instances of indicators: An indicator system contains types of indicators, while indicators that actually measure performance are instances. Especially with regard to *Req. 5*, it would not be satisfactory to neglect such instance level features. For example, a specific indicator type has a 'value' applying to a business process type (e.g., an average over all instances of this business process type); instances of this specific indicator type have a *particularValue*, describing the concrete value of a (projection of) process instance(s), e.g., at a certain time. To address this challenge, we make use of the concept 'intrinsic feature' [6]. An intrinsic feature – marked in the meta model with an 'i' printed white on black – is a type, an attribute or an association defined on meta level, but that reflects a characteristic we associate only to the instance level. Hence, although defined in the meta model this feature is not instantiated at type level but at instance level.

5. CORRESPONDING ARCHITECTURE

The outlined vision – designing and utilizing indicators in an versatile dashboard based on a DSML – requires an architecture for the PMIS that conforms to the requirements identified in Section 2.

First, there is need for a modeling environment that supports the user in designing and maintaining consistent indicator systems and, thus, implements the ScoreML. Figure 6 illustrates the modeling environment in the context of the PMIS architecture. It comprises a modeling editor for the ScoreML as well as – with regard to the integration with an enterprise modeling method – modeling editors for the other modeling languages.

Although the editors are separate, the underlying meta models are integrated (cf. Section 4). Thereby, the modeling environment maintains just a single model ('common model repository'), and the editors act on a defined set of concepts – i.e., parts – of this model. Hence, the 'surrogates' for reference objects in Fig. 5 are replaced by concrete meta types of other (MEMO) languages, like meta types for business processes or resources as indicated in the short example in Figure 2. This facilitates, e.g., cross-model integrity checks, since reference objects in the indicator system model refer-

[2]Note, the meta model at hand is a simplification due to the given restrictions of this paper. The full version can be found at http://openmodels.org/node/190.

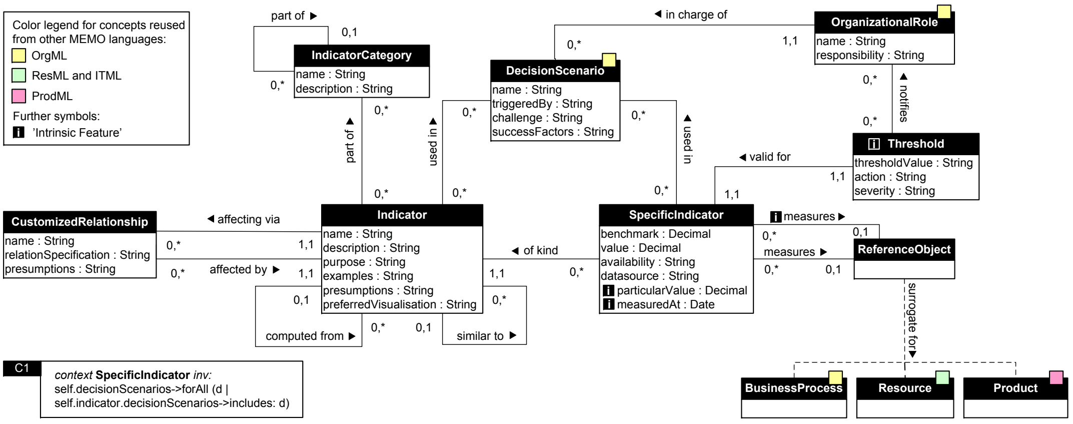


Figure 5: Excerpt of the ScoreML's meta model

ence to existing types in the model repository. Besides this model-based support during *build time*, the architecture also encourages using the models during *run time*. For instance, the language architecture of MEMO allows for navigating between different models (cf. *Req. 4*). Based on associations between concepts – i.e., instantiations of the associations between the corresponding meta types – it is possible to navigate from one model (e.g., an indicator system) to another (e.g., a resource model) by following the association between an indicator type and its reference object (in this case a resource type). The different modeling editors and the depicted language architecture are implemented in a modeling environment – called *MEMO Center NG*.[3]

Second, the tool requires a specific component for visualizing instance values of indicators (cf. *Req. 5*), e.g., by using the typical visualizations such as bar charts or traffic lights. In the architecture, it is represented by the 'dashboard' component. This component can be seen as visualization layer on top of the models (cf. [1, 4]). The proposed architecture poses one pivotal challenge that has to be addressed: The retrieval of instance values that are to be visualized on top of the models requires a connection to the information systems that manage the instance values.

On the one hand, the dashboard component can revert to historical data (e.g., for trend analysis) that are often stored in a data warehouse (DW). This data access requires to enrich an indicator type with a reference to the data source (e.g., tables in the DW) that contains its instance information. An example for a such a reference could be

```
'Select * from Database1.ITCosts
where DateTime between <BeginDate> and <EndDate>'
```

which retrieves the IT costs of a certain time period.

On the other hand, the instance data can be retrieved from operational information systems such as a Workflow Management Systems (WfMS), which contain information about process instances, an Enterprise Resource Planning (ERP) System, which holds information about the business objects such as orders, or a Configuration Management Database (CMDB) that is used to manage information of the IT resources in an enterprise.

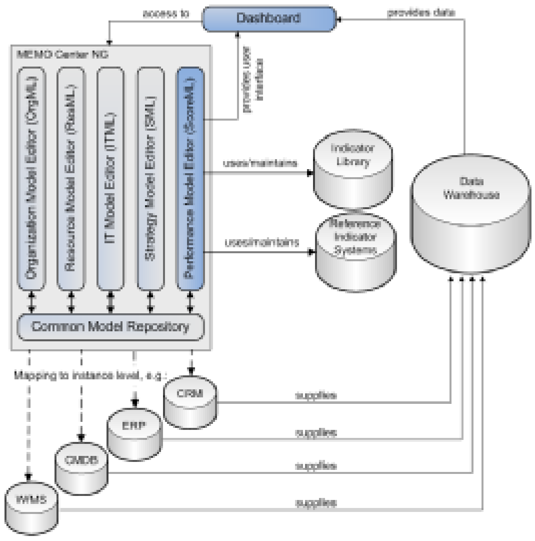


Figure 6: Proposed Software Architecture

The approach is complemented by an extensible set of *reference indicator systems* ('reference models') that are reconstructions of existing indicator systems (e.g., the 'DuPont' indicator system), and an *indicator library* that provides definitions of typical business performance indicators. Both can be loaded into the modeling environment as 'indicator

[3]More details on MEMO Center NG can be found in [6] and at http://www.wi-inf.uni-duisburg-essen.de/fgfrank/memocenter-en

systems building blocks', which serve as a basis for an enterprise specific adaption. More details on the purpose of reference indicator systems and the indicator library can be found in [7].

6. RELATED WORK

There are various information systems to support performance management. However, these tools often focus on the presentation of quantitative data, and they do not provide the user with additional information about indicators or their semantics (like their business context or effect-relations to other indicators; cf. *Req. 1 & 2*). In this regard, data warehouses store data that is extracted, transformed, and loaded from operational information systems to enable various analyses with respect to certain dimensions (like time, region, product) [9]. Thus, data warehouses provide a valuable data source for indicator instances (cf. Fig. 6). Unfortunately, data in data warehouses remain on a low level of semantics. Few approaches exist that try to augment data warehouses with additional context information. For instance, extensions of the EERM [21] or DSMLs [22] exist that allow for modeling dimensions for multi-dimensional structure of data and the navigation between these dimensions, e.g., roll-up or drill-down; hence, they complement our approach with respect to creating and maintaining the data warehouse underlying the PMIS architecture.

There are some commercial tools available (e.g., ARIS[4], ADOscore[5]) that also offer concepts for specifying indicators and – to some extent – allow for assigning them to concepts that represent the business context. However, their language specification is usually not available, and thus the concepts underlying the tool (i.e., the meta models) have to be reconstructed. As far as we can assess those tools, only ADOscore contains a more elaborate conception for indicators with respect to *Req. 1*. Unfortunately, this software is not (fully) integrated with the other ADO-modeling tools (esp., ADONIS), so it still lacks the integration of indicator systems with the business context.

When we developed the indicator modeling method in the context of MEMO, we built upon approaches that focus on indicator modeling (like [18, 23]) and extended those by (1) additional concepts for indicators (e.g., relations) and (2) concepts for the business context (cf. [7] for a more extensive description of these approaches).

7. EVALUATION & FUTURE WORK

In this paper, we outlined the domain-specific modeling language ScoreML for designing and utilizing indicator systems. The language is part of a PMIS that enables using the DSML not only at build-time, but also as versatile front end to instance-level performance data at run-time.

The PMIS consists of several components: a method for indicator modeling, which comprises the DSML and a corresponding process model, a modeling environment that implements the modeling languages, and a software architecture to enable the design and realization of versatile dashboards. In this paper, we focused on the modeling language and its utilization in the context of the envisioned systems architecture (the other parts are introduced in [7]). The design of the language and the corresponding architecture for PMIS were guided by six requirements:

The concepts of the ScoreML have been reconstructed from an existing technical language. Hence, the ScoreML provides its users with an elaborate linguistic structure that guides them with designing transparent and consistent indicator systems (*Req. 1*). By embedding the ScoreML into a method for enterprise modeling that also supports modelling of, e.g., processes, resources, and goals, the indicator models can be enriched with information about the relevant business context (*Req. 2*). Due to the integration of the ScoreML with other modelling languages (here: the family of MEMO languages), different perspectives on indicator systems are supported, e.g., from IT management and business management. This fosters collaboration and communication among the different stakeholders involved (*Req. 3*), as well as it facilitates the analysis of interdependencies between indicators associated with different perspectives (*Req. 4*). We further introduced means in language specification and architecture to integrate the indicator system models with tools that manage corresponding instance level data (*Req. 5*) and that enable the design and utilization of individual performance dashboards (*Req. 6*). Compared to the prevalent practice of creating indicator systems, the ScoreML is promising clear advantages. However, further studies are required to analyze factors such as acceptance and further conditions of successful use in practice.

In our future work we focus on further refining the language. This includes research on the technical integration between the modeling environment and the operational information systems. Also, we will enhance an existing library of reference indicator systems. We will also continue our work on model-driven development of versatile early-warning systems.

Acknowledgments

This research was partially funded by CA Inc. We thank our project partners for their support and valuable comments on preliminary versions of our research results.

8. REFERENCES

[1] S. Buckl, A. M. Ernst, J. Lankes, F. Matthes, C. M. Schweda, and A. Wittenburg. Generating Visualizations of Enterprise Architectures using Model Transformation. *Enterprise Modelling and Information Systems Architectures – An International Journal*, 2(2):3–13, 2007.

[2] W. W. Eckerson. *Performance Dashboards: Measuring, Monitoring, and Managing Your Business.* Wiley & Sons, Hoboken, NJ, 10 2005.

[3] S. Few. Dashboard Design: Taking a Metaphor Too Far. *DMReview. com. March*, 2005.

[4] S. Few. *Information Dashboard Design: The Effective Visual Communication of Data.* O'Reilly, Beijing, 2006.

[5] U. Frank. Multi-Perspective Enterprise Modeling (MEMO): Conceptual Framework and Modeling Languages. In *Proc. of the* 35^{th} *Hawaii International*

[4]http://www.ids-scheer.com
[5]http://www.boc-group.com

Conference on System Sciences (HICSS-35). Honolulu, 2002.

[6] U. Frank. The MEMO Meta Modelling Language (MML) and Language Architecture. ICB-Research Report 24, Institut für Informatik und Wirtschaftsinformatik (ICB), University of Duisburg-Essen, 2008.

[7] U. Frank, D. Heise, H. Kattenstroth, and H. Schauer. Designing and Utilising Business Indicator Systems within Enterprise Models – Outline of a Method. In P. Loos, M. Nüttgens, K. Turowski, and D. Werth, editors, *Modellierung betrieblicher Informationssysteme (MobIS 2008)*, volume 141 of *Lecture Notes in Informatics*, pages 89–106, 2008.

[8] U. Frank and C. Lange. E-MEMO: a method to support the development of customized electronic commerce systems. *Inf. Syst. E-Business Management*, 5(2):93–116, 2007.

[9] W. Inmon. *Building the data warehouse.* John Wiley & Sons, Inc. New York, NY, USA, 1996.

[10] J. Jung. *Entwurf einer Sprache für die Modellierung von Ressourcen im Kontext der Geschäftsprozessmodellierung.* Logos, Berlin, 2007.

[11] R. Kaplan and D. Norton. *Strategy Maps.* Harvard Business School Press, 2003.

[12] R. Kaplan and D. P. Norton. The Balanced Scorecard: Measures That Drive Performance. *Harvard Business Review*, 1992.

[13] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley, New York, 2008.

[14] L. Kirchner. *Eine Methode zur Unterstützung des IT–Managements im Rahmen der Unternehmensmodellierung.* Logos, Berlin, 2008.

[15] R. L. Lynch and K. F. Cross. *Measure Up!: Yardsticks for Continuous Improvement.* Wiley, Blackwell, 1991.

[16] A. D. Neely, M. Gregory, and K. Platts. Performance measurement system design. a literature review and research agenda. *INT J OPER PROD MAN*, 15(4):80–126, 1995.

[17] B. Perrin. Effective Use and Misuse of Performance Measurement. *American Journal of Evaluation*, 19(3):367–379, 1998.

[18] A. Pourshahid, P. Chen, D. Amyot, M. Weiss, and A. Forster. Business Process Monitoring and Alignment: An Approach Based on the User Requirements Notation and Business Intelligence Tools. *10th Workshop of Requirement Engineering.*, pages 80–91, 2007.

[19] V. F. Ridgway. Dysfunctional Consequences of Performance Measurements. *Administrative Science Quarterly*, 1(2):240–247, 1956.

[20] J. M. Rosanas and M. Velilla. The Ethics of Management Control Systems: Developing Technical and Moral Values. *Journal of Business Ethics*, 57(1):83–96, 2005.

[21] C. Sapia, M. Blaschka, G. Höfling, and B. Dinter. Extending the E/R Model for the Multidimensional Paradigm. In *ER '98: Proceedings of the Workshops on Data Warehousing and Data Mining*, pages 105–116, London, UK, 1999. Springer-Verlag.

[22] Y. Teiken and S. Floering. A common meta-model for data analysis based on dsm. In J. Gray, J. Sprinkle, J.-P. Tolvanen, and M. Rossi, editors, *The 8th OOPSLA workshop on domainspecific modeling (DSM)*, 2008.

[23] B. Wetzstein, Z. Ma, and F. Leymann. Towards measuring key performance indicators of semantic business processes. In W. Abramowicz and D. Fense, editors, *BIS*, volume 7 of *LNBIP*, pages 227–238, Innsbruck, Austria, 2008. Springer.

DSML-Aided Development for Mobile P2P Systems

Tihamér Levendovszky, Tamás Mészáros, Péter Ekler, Márk Asztalos
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
H-1111 Budapest
Goldmann György tér 3. IV. em.
Tel.:+36-1-463-2870
{tihamer, mesztam, ekler.peter, asztalos}@aut.bme.hu

ABSTRACT

The proliferation of Mobile P2P systems made a next generation mobile BitTorrent client an appropriate target to compare two different development approaches: the traditional manual coding and domain-specific modeling languages (DSMLs) accompanied by generators. We present two DSMLs for mobile communication modeling, and one for user interface development. We compare the approaches by development time and maintenance, using our modeling and transformation tool Visual Modeling and Transformation System (VMTS).

Categories and Subject Descriptors

D.2.2 [**Software Enginering**]: Design Tools and Techniques – *state diagrams, user interfaces.* D.2.13 Reusable Software – *domain engineering.*

General Terms

Design, Languages

Keywords

Domain Engineering, Methodologies, Graphical environments, Interactive environments, Specialized application languages

1. INTRODUCTION

Mobile Peer-to-Peer technology is a natural demand fueled by the appearance of Smart Phones on the market. The Applied Mobile Research Group at our department did pioneering work in this area. *Symella*, the first *Gnutella* client for Symbian OS, has been downloaded by more than 400,000 users since its first public release in the summer of 2005. *SymTorrrent* is the first *BitTorrent* client for mobile phones. The first free public version was available in 2006 October, as of writing the software has been downloaded by about 300,000 clients. In order to involve mainstream phones into P2P networks, Péter Ekler has developed a *BitTorrent* client named *MobTorrent* for Java ME platform [1]. The original goal was to examine whether mainstream phones are able to run such complex applications. The experiment has met the expectations, and *MobTorrent* became a suitable for communicate with the *BitTorrent* network. The experience stemming from the products made *MobTorrent* an apt environment where we could compare the manual coding and the Domain-Specific Modeling Language (DSML)-aided development.

Having developed the manually coded version, we started with creating domain-specific languages which can be used to describe P2P systems for mobile applications. We identified two main functionality groups where the DSMLs are useful: **processing the protocol messages** and **designing the user interface**. As a DSML platform, we chose the metamodeling and model transformation tool Visual Modeling and Transformation System (VMTS) [2]. In VMTS, we could create the DSMLs, and we could write the model processors that translate the models into Java ME code.

We tried to address the following issues:

- How can Mobile P2P application benefit from DSMLs?
- Does the DSML technology pays off at all in mobile P2P development in time?
- Do the domain-specific models require less maintenance effort?
- Could the DSML approach accelerate the development of the future versions?

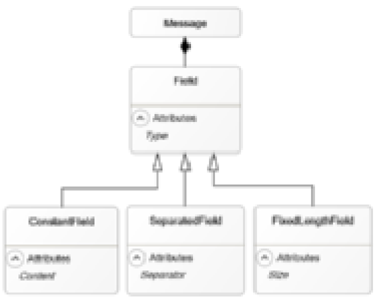
We start with answering the first question by giving an insight of the used DSLs, moreover, we show how we got the facts that underpin the answers.

2. Domain-Specific Languages for Mobile P2P Systems

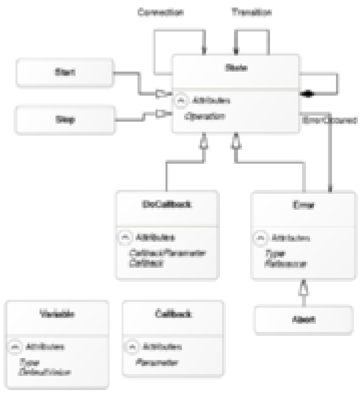
In VMTS, we developed an integrated environment to visually model different aspects of JAVA ME mobile applications, and code generators to turn the models into executable JAVA code. The *Java Resource Editor* DSL is appropriate for the rapid development of the static components of mobile applications, while the *JAVA Network Protocol Designer* can be used to model the static components and the dynamic behavior of simple, message-based network protocols.

2.1 Java ME Network Communication Support

The basics of BitTorrent technology [3] are as follows. In order to download content via *BitTorrent*, firstly we need a very small torrent file. This file contains some meta-data describing the content and the address of at least one central peer called *Tracker*, which manages the traffic. After we have the torrent file, the *BitTorrent* client connects to the *Tracker*, which sends a set of addresses of other peers back to the client. Next the client connects to these addresses and concurrently downloads the content from them via a BitTorrent-specific *peer-wire protocol* [2]. In BitTorrent, we can download the content simultaneously from different peers.



a)



b)

Figure 1 Metamodels for modeling the static and dynamic properties of message-driven state meachines

We developed two DSMLs for modeling the static and dynamic aspects of message-based network protocols, an integrated configuration environment and code generators to support the rapid modeling and implementation of communication through the network. It is capable of describing the peer-wire protocol and its processing logic. Our solution exploits the fact that numerous well-known and widely used network protocols take a message-based approach. This means that the entities communicating with each other use a well-defined language, which consists of exactly identifiable elements with a predefined structure. The *MessageStructure* DSML models the messages (the static components) of such a protocol. Furthermore, the *MessageProcessor* DSML is provided to describe the logic of a protocol. We use hierarchical state machines to define this logic: we can declare the possible incoming messages in a state and the messages to be sent when leaving a state.

With the help of model processors, we can generate a standalone network library, which can be adapted to the user interface or to business logic components. The generated network library provides its services through a unified callback interface. Via this interface it is possible subscribe to numerous events fired by the library during communication.

2.1.1 Modeling messages

Figure 1.a presents the metamodel of the *MessageStructure* DSML. The most important item of this language is the *Message* itself. The message is the unit of the communication of our approach. Each byte sent over the wire has to be the part of a message.

Each message consists of several *Field*s. *Field*s are the building blocks of the messages. *Fields* have a *Type* attribute which corresponds to a simple Java type. Currently *int, byte* and *String* types are supported. We distinguish three different types of fields: *ConstantField, FixedLengthField,* and *SeparatedField.* A *ConstantField* has an additional *Content* attribute which is used to define the exact content of such a field at modeling time. In a protocol where a user ID (e.g.: 123) is sent in the format of *#userid#123*, the *#userid#* part of the message is a *ConstantField*. When reading a message from the network stream, the content of a *ConstantField* must be found at the position defined by the field in the model. Otherwise, the message processing fails. Thus, *ConstantFields* play an elementary role when distinguishing between possible incoming messages in a certain processing state.

FixedLengthFields do not have a predefined content, but a predefined size (*Size* attribute). This means that the field represents a buffer for *Size* pieces of elements of type *Type. The Size* attribute does not have to be a constant value, instead, it can be contained by a field of the same message or global variable (see later), or an aggregated value of those. This means that if we recognize a *FixedLengthField* in a message it is possible that the size of this *FixedLengthField* depends on the already read content of a previous field in this message. *SeparatedField*s do not have a predefined value or size. Their start and end are marked by a character sequence specified in their *Separator* attribute. Reading such a field is finished with reading the value of the *Separator attribute* from the stream. This is a useful feature for textual protocols (e.g FTP or POP3), where the commands are separated with line-break characters.

During code generation, Java classes are created based on the message elements. The contained fields of the messages will correspond to the fields of the Java class. Based on the model and the order of the fields of the messages, we also generate the member methods to read or write the message from or to the network stream. With the help of modeling messages and generating their wrapper classes, our solution completely hides byte-wise network stream operations, and provides an interface based on Java objects to the upper layers of the application.

2.1.1.1 BitTorrent messages

In order to discover and filter the incoming messages described with the *MessageStructure* DSML, we have implemented a message discovery algorithm. After a message is parsed, a callback method is being called which carries the different type of *MessageFields as* parameters. This callback method is used by the *MobTorrent* framework to execute BitTorrent-specific functions such as save the incoming data in a file.

Figure 2 presents the model of the BitTorrent protocol messages. The green fields are the *ConstantFields*, whereas the grey ones are the *FixedLengthFields*. BitTorrent protocol does not use *SeparatedFields*. Usually in every message-based protocol, the messages have a common structure. In the case of BitTorrent we can separate the messages into two parts. The first part contains the *MessageHandshake* (Figure 2) only, which is used during the *peer-wire protocol* to determine whether two peers are compatible with each other. *MessageHandshake* starts with two *ConstantFields* followed by three *FixedLengthFields*. The most

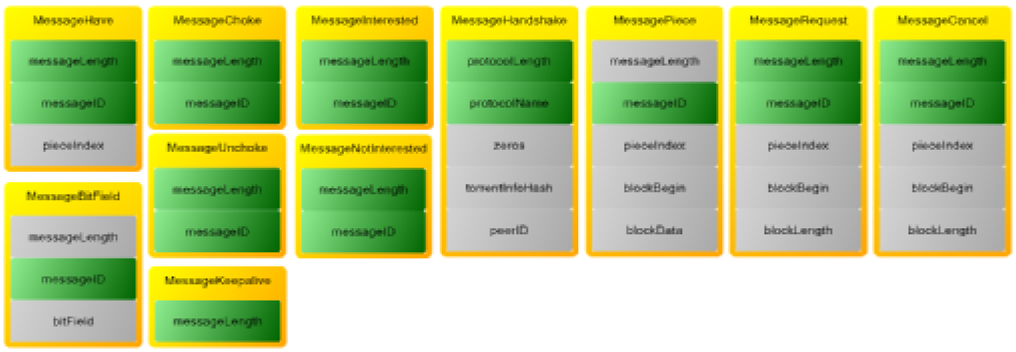


Figure 2 Message objects used by the BitTorrent protocol

important field in the *MessageHandshake* is the *torrentInfoHash*, which is basically the SHA-1 value of the torrent file. This value is used to determine whether the peers are interested in the same content represented by the torrent file.

According to the protocol when peers are exchanging the handshake message, this is the only message which can be accepted, thus, it is easy to discover. After a successful handshake every other message can be sent or received, there are no limitations. However we can see that the structure of these messages is the same. All of them start with a *messageLength* field which defines the length of the message in four bytes. Figure 2 shows that the *messageLength* field is green in all the messages, except for *MessagePiece* and *MessageBitfield*. The length of these two messages depends on the amount of data they carry.

Following the *messageLength*, each message contains a *messageID* field which makes it easy to filter the messages. Only the *MessageKeepAlive* does not have this *messageID* field, because it contains only a *messageLength* constant field.

In the *BitTorrent* protocol, after we have parsed the *messageID* field, we can easily determine which message has arrived, and we can pass the content of the incoming message as the parameters of the callback functions.

2.1.2 *Modeling dynamic behavior*

The core concept of our approach is that that communication layer performs status changes as a consequence of receiving specific messages from the network stream. In addition, we may also instantiate and send network messages during a status change. In our approach, the communication layer can run standalone, and it informs the connecting components of the application through a callback interface about the important events of the communication. The business logic can influence the behavior of the communication layer through the parameters of the layer and by sending messages directly through the network stream. The behavior of the network layer can be modeled with the help of a message-driven state machine.

Figure 1.b presents the metamodel of the *MessageProcessor* DSL. The most elementary item of this state machine is the *State*. There are two special types of states: the *Start* and the *Stop* states. The *Start* state indicates the entry point of the state machine, while the *Stop* state indicates the exit point. As states can be nested (see the containment edge-loop in the metamodel), the start and end states may also be used as the entry/exit point of a sub-state machine. States can be connected with the help of *Transition* edges. A *Transition* edge may trigger the reception of a specific message from the stream: the type of the expected message is defined by the *MessageTypeIn* attribute of the edge, which references an already modeled message. If several outgoing transition edges are connected to the same state, then the transition whose triggered message first arrives will be chosen. If a transition is chosen, the state pointed by its right end will be the next active state. When activating a state, the instruction described in its *Operation* attribute is executed.

An important issue in every message-based protocol is the phase where we have to decide which message has exactly arrived. We have implemented an advanced message handling algorithm especially for the presented *MessageStructure* DSL: in each state of the protocol we have a set of messages which can arrive in the state. Each message knows how many bytes it can consume to process its current (still not read) field. If the size of the current field cannot be determined (in case of a *SeparatedField* or a *FixedLengthField* with variable size) then this message can process only one byte. The key point is that we can increase the efficiency of message parsing, because if we have a set of possible in a state, the minimum amount of bytes to read can be determined. Thus, we do not read the stream by single bytes. Based on the bytes received, we can filter the set of possible messages by the fields with constant values. After a reasonable amount of incoming bytes we can restrict the number of possible messages to one. If neither of the existing transitions is compatible with the data read, we have two possibilities to handle this situation. Either we assume that a *Protocol error* has occurred, and handle the error with a special *ErrorOccured* edge,) or – if no *ErrorOccurred* edges are present and the current state is nested – we let the container state handle the message. We may also attach preconditions to the transitions so that the transition is selected only if the appropriate message arrives, and the condition (*Condition* attribute of the edge) is evaluated to true. In addition, there are two special types of transitions: *non-reading* and *fallback*. A transition is non-reading if it does not trigger any type of incoming message. These transitions are checked only for their *Condition* attribute before choosing them. Therefore they always have priority over the *reading* transitions. *Fallback* transitions (their condition attribute is set to *[fallback]*) are chosen when neither of the other transitions in a state can be selected. This feature is quite analogous to the handling of protocol errors, however, fallback edges are not to handle errors, but it is regular behavior. Recall that a transition may also send a message through the stream. The type of the message sent is defined by the

MessageTypeOut attribute, and the initial value of the fields of the message can be set through the *MessageOutParameter* attributes of the edge.

As already mentioned, the *ErrorOccured* edge is used to handle the errors (either Protocol or I/O) of the network layer. I/O errors occur, when the reading or writing to the stream fails. I/O errors can be handled with *ErrorOccured* edges whose *Type* attribute is set to *I/O.* If none of the outgoing transitions are applicable in a state, and an *ErrorOccured* edge is present whose *Type* is set to *Protocol*, then – regardless of possible container states – we treat this situation as a protocol error. An *ErrorOccured* edge always points to an *Error* state. Error states are special in the sense that a callback method is assigned to each of them on the callback interface. In case of I/O errors, the callback method receives the last exception as well. A special form of *Error* nodes is the *Abort* node, which immediately finishes the execution of the network layer.

The state machine may be customized with *Variables*. Each variable has a name, a type and a default value. Variables can be considered as global parameters, which can be accessed by all states and edges. Variables can be used in the conditions for transitions, during the instantiation of a new message and also in the *Operation* attribute of states. The code generator generates a member variable for each *Variable* node in the model, along with their getter and setter methods. The member variables are initialized with the content of the *DefaultValue* attribute of the corresponding model element.

Recall that a callback method is generated on the callback interface for each error node. Furthermore, a method is generated for each stop state as well. However, one may extend the callback interface arbitrarily, and call its methods from any point of the state machine. For this purpose we have invented the *Callback* node, which symbolizes on method stub on the interface. The generated method can be parameterized with the help of the *Parameter* attribute of that node. Callback methods on the interface can be invoked in two ways: either through a *DoCallback* state, or with the transition edges, as a method invoke can be assigned to each transition. The parameters of the method invoke can be set with the *CallbackParameter* attributes in both cases.

Network connection handling is also modeled with *Connection* edges. Depending on their *Type* property, such an edge either opens or closes the network connection. The target host for the connection is specified by the *Host* attribute. However, the parameters of the connection (connection type, direction, timeout handling etc.) can be customized during code generation.

Figure 3 illustrates the model we have created for the BitTorrent protocol. The yellow boxes represent the global variables of the state machine: (*i) peerAddress* – the address of the peer we are connected to, (ii) *torrentInfoHash* – the hash of the downloaded torrent, (iii) *peerId* – the unique identifier of the connected peer and (iv) *ownPeerId-* our own identifier. The blue boxes with a small yellow lightning denote the callback methods created on the callback interface.

The protocol works as follows. After the start state (1), we call the *Initialize* callback (2) to instruct the framework to perform initialization steps. Then the protocol tries to connect to the target host (the *Connect* edge is parameterized with the *peerAddress* variable). If the connection succeeds, we get to the *Connected* state (3), otherwise an I/O error occurs (4). On error, we perform an *IncreaseErrorCounter* callback, and disconnect the stream. Moving from (3) to (5) a *MessageHandshake* (Figure 2) message is sent to the remote peer. Edges (6) and (7) trigger the answer-*MessageHandshake* message, and check if the parameters of the answer are valid. On an invalid handshake answer, either the *IncreaseErrorCounter* or the *DeletePeer* state will be active, and the communication is closed with the current peer. Edge (8) is a fallback edge meaning that edge (8) is chosen if neither of error transitions (6-7) can be selected. The state *PWConnected* can be considered the default state of the protocol: almost any type of messages can be received at this state (that is why there are so many loop edges around it), and each message arrival performs the appropriate callback invocation. As you can see in Figure 3, the edge parameters (and also other model parameters) can be changed with the help of smart tags. They appear when the mouse is hovered over an item. State *PwConnected* can be left only if a

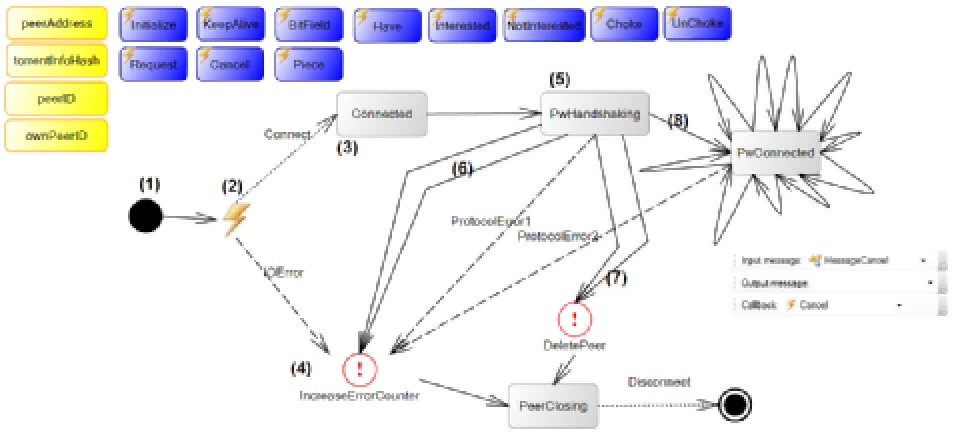


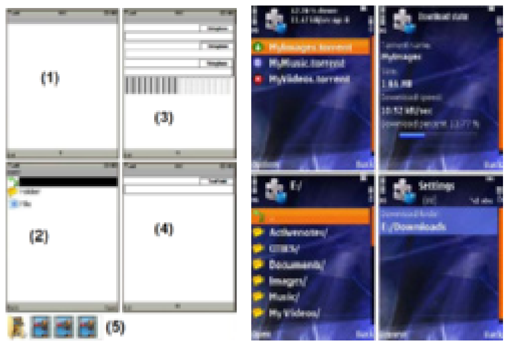
Figure 3 BitTorrent client protocol model

protocol error occurs, or the business logic over the network layer changes the current state.

2.2 Mobile DSL for User Interface Development

Having generated code from the network model and integrated it with the *MobTorrent framework*, we started to work on the user interface of our new mobile BitTorrent client. With UI DSMLs, we can model the static structure of user interfaces, and generate the platform-specific source code according to the models. The UI DSML also has a metamodel, but its detailed explanation is irrelevant. Instead, we are focusing on the models we have created for the BitTorrent application. VMTS supports [4] all the *Screens*, *Commands* and *Controls* available in Java ME both on the modeling and the generator level. Besides the general elements we also provide additional controls which are not part of the Java ME API, but are used in numerous scenarios, such as the *FileSelectDialog*. In P2P applications we usually download multiple contents at the same time and these downloads are displayed in a list where the icon of the list item represents the status (downloading, finished, error, etc.) of the download. With the help of an *ImageList* we can easily access image resources and use them in other components, for example in a List.

In Figure 4, the four screens of the application can be seen both at modeling time (a), and when executing the application on a real hardware (b). Screen (1) is used to present the torrents being processed (*TorrentList*). It is modeled with a simple *JList* item which is replaced with a class derived from Java ME *List* during code generation. Screen (2) is the *FileSelectDialog* itself, with which one can browse for a torrent file to be processed. Screen (3) is used to show the download state of the selected torrent. Screen (3) is built from a *JForm* item, which contains three *StringItems* for presenting the name of the torrent file, the size of the downloaded data, and the actual transfer rate. A *JGauge* element represents a progress bar which shows the progress of the download. Finally, screen (4) is used to modify the application settings such as the download path. It is also based on a *JForm* element, which contains a *JTextField* item. (*JTextField* corresponds to the *TextField* Java ME class).



a)model b)real

Figure 4 UI model of the mobile BitTorent client in VMTS

With the VMTS UI DSML, we can also set the commands (menus) for the screens. The *TorrenList* contains commands for torrent handling such as add torrent file, start download, pause download, and commands responsible for navigating to another screen like *Settings* or *Download state*. Thus, we can also describe the high-level UI logic. The model also contains an *ImageList* (5) with three icons. This list represents the icon set used by Screen (1). Finally, after modeling and generating the network layer and the user interface, one task remains: integrating the generated components with the *MobTorrent* framework. The integration is not supported with visual techniques in the current release of VMTS, the glue-code has to be written manually.

In order to integrate the UI with the *MobTorrent framework* we have applied the *Observer design pattern.* The framework provides an interface which we have to implement in the UI code. This interface contains functions that are called from the framework when the status of the download changes, such as download speed changed, download progress increased. When we initialize the framework we have to set which object implements the observer interface in the UI. By using this observer the framework can notify the UI if something changes and the relevant information can be displayed on the screen of the mobile phone easily.

3. Conclusions

So far we have shown how mobile P2P development can benefit from DSML technology. We found well-separated functionality groups, and supported them by DSMLs and code generators. Table 1 depicts the development times with manual coding and with DSMLs taking one developer into account who had previous experience of this sort of application.

Functionallity	Time with manual coding	Time with DSL
User interface	5 days	2 day
Peer network connection	8 days	1 day
Peer-wire protocol	6 days	1 day
Message handling	10 days	2 days

Table 1. Development time with and without DSMLs

Additionally, there were functions, which we did not support with DSMLs. These required the following amount of time:

- File and database handling: 8 days
- BitTorrent specific functions: 13 days
- Tracker communication: 5 days
- Download for other clients: 8 days

The DSML infrastructure, i.e. the languages and the generators, is developed by an engineer with extensive DSML and tool experience. The time spent per person is the following:

- *MessageStructure* and *MessageProcessor* DSMLs: 4 days
- *MessageStructure* and *MessageProcessor* generators: 5 days
- UI DSML: 9 days
- UI generator: 10 days

So the development effort for the functions supported by DSMLs is as follows:

- With DSMLs: 6 days
- Without DSMLs: 29 days

The development time without the time for the DSML development:

- With DSMLs: 40 days
- Without DSMLs: 63 days

Including the DSML infrastructure development:

- With DSMLs: 68 days
- Without DSMLs: 63 days

These numbers shed a light on the fact that DSML technology is a generative technique: a generator is much harder to develop than the generated code once. Therefore, the more times you run the generator, the more the DSML approach pays off. From the second time on, the DSML and generator development does not appear as an additional cost. Our intention was to support UI changes, protocol changes and updates in the forthcoming version of a P2P application. That is why we support these functions and not others. As a matter of fact, our figures look better: we inherited the UI DSML from another project targeting cross platform UI development. Thus, we can subtract it from the total, and we have 47 days for the *MobTorrent* project.

As long as only the models need to be modified, DSMLs increase the maintainability. If the generator must also be modified, the necessary effort can arbitrarily increase. We expect that we need to modify the models for the next versions because of the generality of the DSMLs, and subtle generator modifications if the Java ME UI changes. These DSMLs can be reused for any Java ME mobile development where UI or network support is required, but the approach is not limited to the Java ME platform, since it can be extended to other platforms by modifying the code generators. The proposed case study can be used as well in other solutions where BitTorrent technology is used for content distribution. Since our department is involved in developing such applications on a regular basis, we have a rational expectation to have the return of our investment in the DSMLs and the supporting generators as it happened in the case of the UI models.

We tested the generated code, and decided to include it in the first release of *MobTorrent*. Thus, *MobTorrent* is expected to be publicly available in January 2010 on its website, as the first mobile P2P client developed with the extensive help of DSMLs.

ACKNOWLEDGMENTS

The found of “Mobile Innovation Centre” has supported, in part, the activities described in this paper. Infragistics has supported, in part the activities described in this paper.

4. REFERENCES

[1] P. Ekler, J. K. Nurminen, A. J. Kiss “Experiences of implementing BitTorrent on Java ME platform”, CCNC’08. 1st IEEE International Peer-to-Peer for Handheld Devices Workshop, pp. 1154-1158, 2008, USA

[2] Visual Modeling and Transformation System Website: http://vmts.aut.bme.hu

[3] BitTorrent specification, Oct. 13, 2008. [Online]. http://wiki.theory.org/BitTorrentSpecification

[4] I. Madari, L. Lengyel, T. Levendovszky “Modeling the User Interface of Mobile Devices with DSLs”, Proc. of the Computational Intelligence and Informatics 8th International Symposium of Hungarian Researchers, pp. 583-589, 2007, Hungary

MobiDSL - a Domain Specific Language for Mobile Web Applications

: developing applications for mobile platform without web programming

Ankita Arvind Kejriwal
Birla Institute of Technology and Science, Pilani.
Goa Campus, India
kejriwal.aa@gmail.com

Mangesh Bedekar
Birla Institute of Technology and Science, Pilani,
Goa Campus, India
bedekar@bits-goa.ac.in

Abstract

The enormous potential of mobile web as an information appliance presents all organizations an urgent need and a compelling reason to not only create mobile specific versions of certain parts of their current systems, but also develop new mobile web applications to derive maximum benefit from this medium. However, as mobile web applications are generally being built using the same web engineering methodologies and tools which are used for building desktop web applications, organizations around the world require significant resources, making it difficult for many to quickly build these applications. In this paper, we describe our aim to mitigate this problem by using a Domain Specific Modeling (DSM) based approach. We explain MobiDSL - a Domain Specific Language (DSL) for modeling Mobile Web Applications and show how it can enable system designers and analysts to easily define an application's specification at a very high level of abstraction without any web programming. We also explain how a Virtual Machine (VM) is used to execute MobiDSL models, which helps to radically simplify the testing, deployment and life cycle management of such mobile web applications.

1. Introduction

Mobile Web refers to web content (static or dynamic) which is specifically designed to be accessed on mobile devices via a browser [2]. The mobile web is emerging as an information medium whose reach is projected to surpass all other mass media (including the print, cinema, radio, television and desktop internet) as mobile devices have the advantage of being personal, portable, always on and connected. It presents us with a massive opportunity to provide information and e-services to entire human population, not just in developed countries but also in developing and under-developed regions of the world. It can be the best enabling mechanism to empower billions of people with information and to bridge the digital divide.

While it is possible to access standard web content on some mobile devices, the user experience is often less than satisfactory primarily due to smaller screen size and lower bandwidth compared to a desktop or laptop. Therefore, Mobile Web Best Practices (MWBP) [2] specifies practices for delivering web content to mobile devices. These recommendations refer to the 'content' and not to the processes by which the content is created or delivered. We find emergence of following trends in creation and delivery of web content to mobile devices.

1. The static web content is either being re-developed or being transformed at runtime by using server side software such as Instant Mobilizer [4].
2. The dynamic web content is being provided by a new class of re-engineered, light-weight mobile specific versions of corresponding desktop web applications. Some of the notable examples are dynamic content sites such as BBC and ESPN; and dynamic applications such as Gmail, Facebook, and Twitter.

For the purposes of this paper, we define *Mobile Web Application* as "a web application specifically designed to deliver dynamic information from the database and provide simple transactional services to mobile devices via a browser".

The rapidly growing popularity and success of many mobile web applications has demonstrated the enormous potential of mobile web as an information appliance. Organizations are faced with an urgent need as well as a compelling reason to not only create mobile specific versions of certain parts of their current systems (legacy, client-server or web applications) but also to develop new mobile web applications which derive maximum benefit from this medium.

Though mobile web applications are simpler compared to desktop web applications, they are generally being built using the same web engineering methodologies and tools which are used for building desktop web applications. Thus, organizations around the world would require significant resources to design, develop, test and deploy mobile specific versions of their current as well as new applications. As a result, it might be difficult for many to quickly develop these applications.

A light-weight development methodology for developing mobile web applications which could supplement and co-exist with whatever methodology an organization might be using for their core systems could help address the aforesaid issue. This methodology would also have to be simple enough so that it can be used directly by system designers and analysts to develop such applications.

We attempted to find a solution using the DSM based approach. We analyzed the mobile web domain to identify various constructs and designed a simple, compact and extensible DSL called MobiDSL for defining mobile web applications. Though DSM based approaches usually involve generation of code which can be compiled and run, we chose to develop a Virtual Machine (VM) to execute MobiDSL models due to various benefits which are described later. Our approach aims to provide a reasonably sound framework for rapid prototyping, development, testing, deployment and life-cycle management of mobile web applications without the need of any web programming or provisioning of any special middleware.

The rest of the paper is structured as follows. Section 2 describes the background and related work. Section 3 describes our approach, objectives and the VM. It also explains MobiDSL with a few succinct examples and finally provides a brief overview of MobiDSL's meta-model. In Section 4, we discuss various issues and describe the benefits of interpretive approach over generative approach. Section 5 presents a summary of contributions and future work.

2. Background and Related Work

Mobile internet access began with the Wireless Access Protocol (WAP), wherein the pages were composed in Wireless Markup Language (WML). The advent of WAP 2.0 permitted use of XHTML markup with end-to-end HTTP. To assist development community, W3C has launched *Mobile Web Initiative* [1] and published standards such as Mobile Web Best Practices (MWBP), XHTML for mobile (XHTML-MP), Cascading Style Sheet for mobile (CSS-MP), and MobileOK Basic Tests.

Mobile Web applications are generally being built using the same Web Engineering methodologies which are used for building regular web applications. Most methodologies follow a standard three tier architecture as described below:

a) A database tier – which stores the database and runs the database server
b) An application tier – which runs the actual application logic. The application server receives the user inputs from the client device via HTTP (over TCP/IP) and can use various API's to interface with HTTP server. It can interface with database server using various database API's to retrieve information or update transactions.
c) The client tier – which runs the application using a web browser. The browser renders pages composed in HTML. CSS is used to specify visual aspects rather than specifying them in HTML for each hypertext element. The client tier can also use technologies such as DHTML (to dynamically construct HTML fragments), JavaScript (to handle events) and Ajax (to asynchronously access the application tier).

There is a plethora of Web Engineering methodologies and frameworks ranging from scripting technologies such as ASP, JSP and PHP (to name a few) to enterprise class web technologies such as J2EE. There are also sophisticated web modeling based frameworks such as HDM, WAE, WSDM, UWE and WebML. The scripting based systems provide API for interfacing with HTTP server and Database Server and allow the developer to hand code the programs. Many systems also provide tools and templates to automate generation of boiler-plate code. The J2EE is a sophisticated enterprise class technology which divides application server into partitions, which can be run on different machines to achieve high degree of scalability. The modeling based frameworks provide facilities for various facets of modeling such as Content Modeling, Hypertext Modeling, Presentation Modeling and Customization Modeling. Some of them also provide tool support to partially or fully generate application code. There are also some XML based frameworks employing XML transformation (XSLT) such as Apache Cocoon.

However, creating a robust and scalable mobile web application using any of the above methodologies is not a trivial task. Many researchers have advocated use of *Domain Specific Language based frameworks* to develop robust, reliable and secure programs in a cost effective manner by using high level abstractions. We were inspired by the reported success of DSM approach in many application areas [5, 6] and the use of DSL paradigm in web engineering by some researchers such Nunes et al. [7], Martin et al. [8], E. Visser [9] and Ceri et al. [10].

We were particularly influenced by *Web Modeling Language (WebML)* proposed by Ceri et al. [10], which stands out for enabling high level abstraction of various models such as structural model, composition model, navigation model, presentation model, and customization model. However, whereas WebML's meta-model is very elaborate, we have combined the models into one simple model. Moreover, whereas WebML's tool generates the application code, we have implemented a VM to execute the model. Such a concept of executable DSL models has already been demonstrated in ModelTalk [11].

3. MobiDSL

3.1 Overview of Mobile Web Applications

The Mobile Web Applications aim at providing a compelling user experience. We find evolution of new design philosophy for mobile web, whose central theme is based on simple and minimalistic design concept [3].

A mobile web application consists of various web pages, which can be broadly classified as Home Page, Query Pages and Transaction Pages. These pages are hyperlinked to each other to enable navigation thorough the system. The links can be either Contextual Links (which pass certain contextual information while navigating to another page) or Non-Contextual Links.

The Home Page typically provides certain static choices (menu options). It may contain an authentication section to allow only authorized users to access the application. It can also contain choices created dynamically based on certain information in the database.

The Query Pages enable the users to access information from database based on some selection criteria. The selection criteria can be either based on search parameter inputs defined in the same page or query string passed to the page by a contextual link. The Query Pages can either display several records (List View) or a single record (Record View).

The Transaction Pages enable the mobile users to update simple transactions on the database. These transactions typically allow Create, Update and Delete facility on records in a specified table. They allow elementary validations to be performed on the data entered by users.

Some mobile applications also require special features such as Access Control for restricting access to some pages to a certain group of users based on their user role. Some applications might require personalization features, wherein user's preferences such as presentation skins, accessibility preferences and favorites are stored in a database table, and web pages are created accordingly.

The mobile web pages typically have different sections as:

- Page Header - which may consist of Branding (Logo, Name), Title and Navigation Tree
- Body - which can consist of menu options, search request section, query view section or transaction entry section
- Page Footer - which may consist of Branding, certain links or other relevant information

We find that Mobile Web Applications differ from regular Web Applications in several ways which are summarized below:

1. Device limitations - screen/keypad size and bandwidth
2. Different Usage pattern
3. Minimalistic design with Simple Layouts (no frames or nested tables or complex layouts)
4. No processing on client devices using DHTML, JavaScript or Ajax, as this would render the application unusable on a majority of mobile devices.
5. Lesser functional expectations

3.2 Our Approach

The simple nature of mobile web application presents an opportunity to create new lightweight frameworks suitable for this emerging medium. Our approach to creating a new framework for mobile web applications was based on the promise of DSM approach and on the premise that the mobile web domain can be analyzed to identify the core requirements and various design elements, based on which a domain specific language (DSL) can be designed. DSL is “a high-level software implementation language that supports concepts and abstractions that are related to a particular (application) domain” [9].

The driving factor for identification of language constructs was primarily based on “Domain Expert’s and Developer’s Concepts” approach [5]. Based on our earlier work in developing mobile web applications using scripting methodologies, we identified distinct features and recurring themes in these applications. We also discussed with an industry expert to understand the domain expert’s concepts on creating mobile versions of current systems without any programming. We were also partly led by “look and feel” approach [5] as far as the user interface (hypertext) was concerned.

We have designed a simple and concise textual DSL suitable for developing mobile web applications called MobiDSL, which enables a developer to define the specifications for each page at a very high level of abstraction without requiring any knowledge of web programming. We have used XML as the Meta Language as XML markup is simple, flexible and easily understood. MobiDSL allows us to define:

1. Page structure
2. Hypertext (text, widgets and links) in each section
3. SQL for data retrieval
4. Query result presentation
5. Validations and other business logic for transactions

3.3 Objectives of our framework

Our framework aims to:

1. Simplify mobile web application development by allowing designers, analysts and programmers to define application at a high level of abstraction
2. Allow the application to be run on most mobile devices by carrying out all processing and validations on server and using only XHTML-MP (with CSS-MP) on client device
3. Simplify deployment by adopting a simple three tier distributed RESTFUL architecture, which employs only basic protocols (such as HTTP) and basic API’s (such as CGI and Pass Through SQL)
4. Provide scalability by allowing multiple application servers wherein a request could be serviced by any of them.
5. Optimize processing by caching to retrieve result sets of recently executed queries
6. Avoid middleware complexity normally seen in many high end web applications

3.4 The Virtual Machine

The Virtual Machine (VM) runs the mobile web application as embodied in the MobiDSL Model. It is implemented as a stateless, distributed and scalable Application Server. A schematic view of the MobiDSL VM deployment is shown in figure 1. VM uses the configuration information in sys.xml file to connect to Database Server and Memcached. On one side, the VM interfaces with HTTP server encapsulating the CGI API. On the other side, it interfaces with database and encapsulates the database access API. When the VM receives client input it identifies the page to be delivered based on page identifier in query string. It parses the corresponding MobiDSL model (<pageid.xml>), runs database queries based on search request and constructs the response XHTML page (with embedded mvm.css) as specified in model. In a nutshell, it handles complete server side processing for the mobile web application (encapsulating various web engineering technologies) based merely on the application’s MobiDSL model. The VM also manages Sessions, Pagination, and a Navigation tree. The response pages created by VM are compliant with mobileOK Basic Tests.

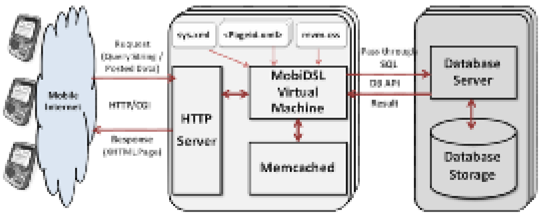


Figure 1. A Schematic View of MobiDSL VM Deployment

We have implemented this system on classic Linux-Apache-MySQL-PHP platform. The implementation follows a distributed three tier model, allowing any number of application servers to be connected to database server. A server based session management stores the session data in Memcached on server side, allowing that information to be retrieved from any application server. This makes each request-response cycle completely stateless, making it possible for each user interaction to be serviced by a different application server, making the system scalable and reliable. The VM also caches the result set of recently executed queries to optimize processing.

3.5 Sample Application

To understand MobiDSL, let us consider an example of a Mobile Web Pharma Sales Force Application used by Medical Sales Representatives (MSR). One of the tasks of the MSR is to visit various physicians (in the towns assigned to her) periodically to brief them about the company’s products. The application enables an MSR to view the list of physicians and details of previous visits. It also enables her to record details of a new visit on-line. A simplified content model of relevant parts of the application is shown in figure 2.

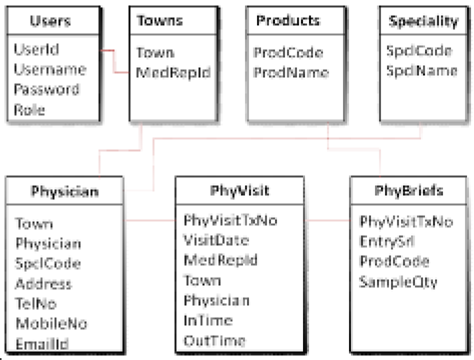
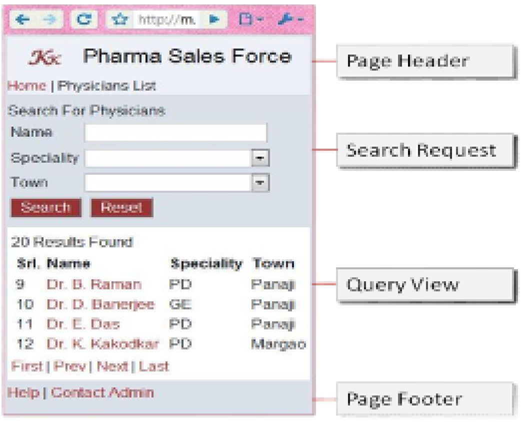


Figure 2. Content Model of Pharma Sales Force Application

3.6 The Physicians List Query

We first consider the physicians list query page/screen shown in figure 3. This screen enables the MSR to view a list of all physicians in the towns assigned to her. Moreover, it also allows her to optionally search based on part of name, specialty code or town.



20 Results Found

Srl.	Name	Speciality	Town
9	Dr. B. Raman	PD	Panaji
10	Dr. D. Banerjee	GE	Panaji
11	Dr. E. Das	PD	Panaji
12	Dr. K. Kakodkar	PD	Margao

Figure 3. The Physicians List Page

The specification for this page as defined in MobiDSL is shown in figure 4. The root element `page` identifies the page and contains other elements such as `pageheader`, `searchrequest`, `queryview` and `pagefooter` corresponding to each section in the page.

The MobiDSL code for Page Header specifies a static image (stored on server in directory relative to location of VM), the page header title, a link back to the home page and the current page's title. Most of these specifications are self-explanatory.

The code for Search Request section specifies a text line, input widgets for search criterion and Submit and Reset buttons. The SQL statements are used to specify the options for input widgets such as `Specialty` (all specialties to be shown) and `Town` (only those towns which are assigned to MSR to be shown).

The code for Query View section specifies the SQL to be executed for retrieving required data from database based on the search criterion. Here, the SQL specifies the physician's list by using a join of the `Physician` table with the `Towns` table (for only those towns assigned to a MSR), and further filtering records based on search criterion. The search inputs submitted by the client device are stored in variables with names corresponding to input widget name prefixed by $. Using caret (^) in expressions such as `a.town=$town^` implies that if the input stored in `$town` is empty, the expression is to be ignored (reduced to logical true by VM). The data retrieved from the database using SQL can be presented in various layouts such as `para`, `dualcol` and `table`. The `resultrecord` element specifies presentation for each record and can contain one or more `resultcol` elements (data columns). A `resultcol` element can also specify a link which is invoked when user clicks on that column's value. It can pass that value (as well as any other value using `passvalues` attribute) as query string to the resource pointed to by it.

```
<?xml version='1.0' standalone='yes'?>
<page id="phylist">
  <pageheader>
    <image src="images/psf.gif" nobreak="true"/>
    <text class="title">Pharma Sales Force</text>
    <text href="mvm.php?pageid=home"
          nobreak="true">Home</text>
    <text expr="' | '" nobreak="true"/>
    <text>Physicians List</text>
  </pageheader>
  <searchrequest>
    <text>Search for Physicians</text>
    <input type="text" label="Name" name="physician">
    <input type="select" label="Speciality"
       name="spclcode"
       optionsql="select spclCode from specialty"/>
    <input type="select" label="Town" name="town"
       optionsql="select town from towns where
                  medrepid=$_userid"/>
    <submit label="Submit" />
    <reset label="Reset" />
  </searchrequest>
  <queryview layout="table" recordsperpage="4">
    <sql>select a.* from physician a, towns b
       where a.town=b.town and b.medrepid=$_userid
       and match(a.physician) against ($physician^)
       and a.spclcode=$spclcode^ and a..town=$town^
       order by physician</sql>
    <text expr=" $_reccount . 'Results Found' "/>
    <resultrecord>
      <resultcol label="Name" sqlcol="physician"
         href="mvm.php?pageid=phydet"
         passvalues="town;physician"/>
      <resultcol label="Speciality"
         sqlcol="spclcode"/>
      <resultcol label="Town" sqlcol="town" />
    </resultrecord>
  </queryview>
  <!-- pagefooter code omitted -->
</page>
```

Figure 4.The Specification for Physicians List Page in MobiDSL

3.7 Key Concepts

Sequence of Events and Processing. Though some MobiDSL element tags and properties might appear to be similar to HTML tags, they are not HTML tags. Whereas HTML is processed by the browser on the client device, MobiDSL code is processed by the VM (on Server) to handle client device inputs or create HTML output pages accordingly. The following points summarize the sequence of events and processing for the previous query page.

1. When this page is requested (as mvm.php?pageid=phylist) for the first time, the VM creates initial XHTML page with a list of all physicians in all towns assigned to the MSR (as all search inputs are empty at that time) in following steps:
   a) VM reads MobiDSL code for target page (`phylist`) and parses it. It saves parsed DSL in cache for re-use.
   b) VM begins to construct the XHTML page by generating the `<head>` section with page title and embedding the CSS file as inline `<style>`.
   c) VM then generates HTML for Page Header.
   d) Then, VM generates HTML for Search Request section. If the section contains any Select widgets, it populates options for these widgets from result sets of corresponding `optionsql`. The HTML for this section is embedded in `<form>` tag. VM also embeds control information like `pageid` & `sessionid` as hidden fields.
   e) To create HTML for Query View section, the VM first prepares the SQL statement (by substituting the values of variables used in SQL) and submits it to database server. The VM fetches the result set and constructs HTML for presenting the data in the specified manner.
   f) Then, VM generates the HTML for Page Footer section.
   g) It finally sends the fully constructed XHTML-MP page

with embedded CSS to the client device.

2. The browser on the client device loads the XHTML page. The user can see list of first four physicians. The user can use `Next` link to request the server to send a page with next set of records. The user can also select the link associated with physician name to navigate to Physician's Details Page. The user can also enter values for some search criteria and press `Submit` to request the server for a filtered set of data.
3. When the user submits the search criteria, it is posted to the VM. The VM retrieves control information such as `pageid` and `sessionid` from the posted data. It loads the parsed MobiDSL code from cache based on `pageid`. It retrieves search inputs from posted data and cleans it to guard against SQL injection attacks. The VM then prepares the SQL statement as explained earlier and submits it to the database server. It then fetches the result set and re-constructs the XHTML page and sends it back to the client device.
4. When the user selects any of the pagination link (`First`, `Prev`, `Next` or `Last`), a query string is sent to VM with appropriate information. The VM receives the query string and processes it in a manner similar to (3). The VM fetches the result set from cache and reconstructs the XHTML page with relevant records.

Variables MobiDSL gives us the flexibility to refer to various variables in SQL statements or other expressions. These variables are created and managed by VM automatically in their respective context; and can be referred to in the specifications in appropriate contexts. In a nutshell, these variables are:

1. Authentication variables: These are variables like $_userid and $_userrole. They have global scope.
2. Query String Variables: Name-value pair collections from a contextual link are stored in variables with corresponding names prefixed by $. For example, if a query string is `field1=value1&field2=value2`, then two variables, `$field1` and `$field2` will be created. These variables are in scope of the page in which they are received.
3. Inputs Fields in a Page: The values received as posted data from a page are stored in variables with corresponding names of input fields prefixed by $. These variables are in scope of the page in which they are received.
4. SQL Query Result Set Control Variable: the `$_reccount` variable gives the count of records retrieved by query. It is in scope of `queryview` section.
5. SQL Query Result Record Level Variables: a) `$_currec` which gives the current record number. b) Result Columns for current record stored in variables with the column names prefixed by `$_sqlres_`. These variables are in scope of `resultrecord` specifications.

Expressions. In MobiDSL specifications, we can use expressions to define some attributes of various elements. These expressions can be any PHP expression comprising any PHP functions (in-built functions, library functions or user-defined functions) and any of the variables available in the given context. The ability to use expressions in several attributes such as following in the MobiDSL specification enables the developer to define various requirements with ease:

1. `expr` in text and `resultcol` elements can be used to define an expression to display required string or value
2. `hrefexpr` in image, text or `resultcol` elements to define an expression for a link associated with that element
3. various attributes such as `defvalexpr`, `disableifexpr`, `validexpr` etc. for Input field elements in a transaction

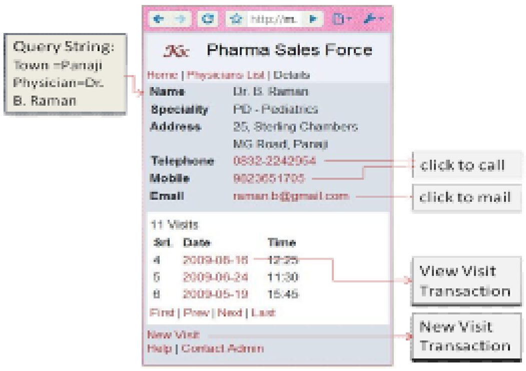
It is to be noted that the developer can provide additional functionality by developing application specific PHP functions which can be used in expressions. This feature makes MobiDSL reasonably extensible while keeping its core grammar to a minimum.

Pass Through SQL. MobiDSL allows us to define a pass through ANSI SQL to retrieve any data from the database. The SQL can be of any complexity involving any number of tables and can contain MobiDSL variables. The MobiDSL VM prepares the SQL statement by substituting the values of variables and then submits it to the database server in a pass through fashion. The SQL is executed as such by the database server and the result set is sent back to the VM. The SQL can be used in following contexts in the MobiDSL specification:

1. In `queryview` section to retrieve the data to be presented.
2. In `optionsql` attribute for select input widgets to populate the select options.
3. In `pageheader` and `pagefooter` to retrieve required control information from the database.

3.8 The Physicians Details Query.

We now consider the physicians details query page/screen shown in figure 5. This screen enables the MSR to not only view contact details, but also call or email the physician using a single click. It shows a list of previous visits to the physician and allows the MSR to view any previous visit transaction. It also allows her to initiate a new Visit Transaction.



Pharma Sales Force

Home | Physicians List | Details

Name	Dr. B. Raman
Speciality	PD - Pediatrics
Address	25, Sterling Chambers
	MG Road, Panaji
Telephone	0832-2242954
Mobile	9823651705
Email	raman.b@gmail.com

11 Visits

Srl.	Date	Time
4	2009-06-16	12:25
5	2009-06-24	11:30
6	2009-05-19	15:45

First | Prev | Next | Last

New Visit

Help | Contact Admin

Figure 5. The Physicians Details Query Page

The specification is shown in figure 6. The `queryparams` specify query string parameters received by this page. The page contains two `queryview` sections:

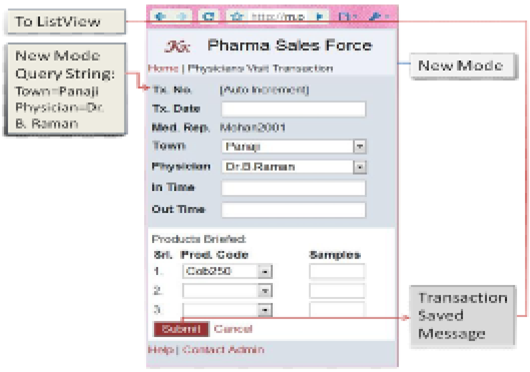
1. Physician's Contact Details (Single Record View) - Here, the SQL specifies the desired selection from the physician table using query string parameters. The layout is set to be a two column layout where the first column shows label and second column shows data. The Telephone, Mobile and Email values are rendered as links using `hrefexpr` which allows us to specify PHP expression for defining the link.
2. Visit Details (List View) - The SQL specifies selection from `phyvisit` table. The layout is set as table showing the visit date and time. The visit date is rendered as a link to let the user to navigate to Visit Transaction in view mode.

```
<?xml version='1.0' standalone='yes'?>
<page id="phydet">
  <queryparams>
    <param name="town"/>
    <param name="physician"/>
  </queryparams>
  <!-- pageheader code omitted -->
  <queryview multirecord="false" layout="dualcol">
    <sql>select a.*,b.spclname
         from physician a,Speciality b where
         a.spclcode=b.spclcode and a.town=$town
         and a.physician=$physician</sql>
    <resultrecord>
     <resultcol label="Name" sqlcol="physician"/>
     <resultcol label="Speciality"
     expr="$_sqlres_spclcode.'-'.$_sqlres_spclname”/>
     <resultcol label="Address" sqlcol="address"/>
     <resultcol label="Telephone" sqlcol="telno"
        hrefexpr=" 'tel:'. $_sqlres_telno "/>
     <resultcol label="Mobile" sqlcol="telno"
        hrefexpr=" 'tel:'. $_sqlres_mobileno "/>
     <resultcol label="Email" sqlcol="telno"
        hrefexpr=" 'mailto:'. $_sqlres_emailid "/>
    </resultrecord>
  </queryview>
  <queryview layout="table" recordsperpage="5">
    <sql>select * from phyvisit where
         town=$town and physician=$physician
         order by visitdate desc</sql>
    <text expr=" $_reccount . 'Visits' " />
    <resultrecord>
      <resultcol label="Date" sqlcol="visitdate"
         hrefexpr="'mvm.php?pageid=phyvisit'.
                   '&amp;txmode=view'"
         passvalues="phyvisittxno"/>
      <resultcol label="Time" sqlcol="intime" />
    </resultrecord>
  </queryview>
  <!-- pagefooter code omitted -->
</page>
```

Figure 6. The Specification for Physicians List Query Page

3.9 The Physicians Visit Transaction

Now we consider Physicians Visit Transaction page/screen as shown in figure 7. This screen enables the MSR to enter details of her visit including the products that she briefed to a physician.


To ListView
New Mode Query String: Town=Panaji Physician=Dr. B. Raman
New Mode
Transaction Saved Message
Pharma Sales Force
Home | Physicians Visit Transaction
Submit
Cancel
Help | Contact Admin

Tx. No. [Auto Increment]
Tx. Date
Med. Rep. Mohan2001
Town Panaji
Physician Dr.B.Raman
In Time
Out Time

Products Briefed:

Srl.	Prod. Code	Samples
1.	Cob250	
2.		
3.		

Figure 7. The Physicians Visit Transaction Page

The specification for this page is shown in figure 8. As before, the `queryparams` specify query string parameters received by this page. While the query parameter `phyvisittxno` is passed to load the transaction in View Mode, the `town` and `physician` parameters are passed in New Mode to serve as default values for these fields. This page contains a `simpletxn` section which contains two transaction blocks (identified by `txnblock`):

1. The first transaction block corresponds to `phyvisit` table. This block is defined as a parent block with single record set in dual column layout. While the `tablekeys` specify primary keys of the table, `loadkeys` define corresponding variables whose values should be used to retrieve the transaction in View or Edit Mode. The block contains several fields, the first of which is Tx. No, whose datatype is defined as `autoincr` (value of field to be generated by database server while saving the record). The next field is Tx. Date, which has default value of current date. The Med. Rep. field is a protected field with a default value of MSR's `userid`. Then, we have more input fields whose specifications are self-explanatory.
2. The second transaction block corresponds to `phybriefs` table. This block is defined as a child block with multiple records set in tabular layout. The first field here is Prod. Code which has a foreign key validation against `products` table. The second field is `Samples` which is optional.

While the transactional model uses *Controls* such as `Submit` button and `Cancel` link in New Mode, it uses `Edit` link, `Delete` link and `Back` link in View Mode. The behavior of these controls is fixed and the developer can only specify alternate labels for these controls.

```
<?xml version='1.0' standalone='yes'?>
<page id="phyvisit">
  <queryparams>
    <param name="town"/>
    <param name="physician"/>
    <param name="phyvisittxno"/>
  </queryparams>
  <!-- pageheader code omitted -->
  <simpletxn>
   <txnblock blocktype="parent" multirecord="false"
       layout="dualcol" tablename="phyvisit"
       tablekeys="phyvisittxno"
       loadkeys="$phyvisittxno" />
     <input type="text" label="Tx. No."
         name="phyvisittxno" datatype="autoincr"
         disableifexpr="1"/>
     <input type="text" label="Tx. Date"
         name="visitdate" datatype="date"
         defvalexpr="date('Y-m-d')"/>
     <input type="text" label="Med. Rep."
         name="medrepid" datatype="char" size="20"
         defvalexpr="$_userid" disableifexpr="1"/>
     <input type="text" label="Town"
         name="town" datatype="char" size="20"
         defvalexpr="$town" disableifexpr="1"/>
     <input type="text" label="Physician"
         name="physician" datatype="char" size="30"
         defvalexpr="$physician" disableifexpr="1"/>
     <input type="text" label="In Time" name="intime"
         datatype="time" size="5"/>
     <input type="text" label="Out Time"
         name="outtime" datatype="time" size="5"
         valdexpr1="$outtime > $intime"
         valdmsg1="OutTime must be morethan InTime"/>
   </txnblock>
   <txnblock blocktype="child" multirecord="true"
       layout="table" tablename="phybriefs"
       tablekeys="phyvisittxno"
       loadkeys="$phyvisittxno" />
     <title>Products Briefed</title>
     <input type="select" label="Prod. Code"
         name="prodcode" datatype="char" size="10"
         optionsql="select prodcode from products"
         fkeytable="products"
         fkeytablefields="prodcode"
         fkeyvaluefields="$prodcode"
         fkeyerrmsg="Invalid Product"/>
     <input type="text" label="Samples"
         name="sampleqty" datatype="num" size="2"
         required=”false”/>
   </txnblock>
  </simpletxn>
  <!-- pagefooter code omitted -->
</page>
```

Figure 8. The Specification for Physicians Visit Transaction

3.10 MobiDSL Metamodel

We can see that MobiDSL has very high expressive power as it allows us to specify page structure, presentation (static/dynamic text, widgets), navigation, data retrieval, data formatting and transactional logic at almost same level of expression as required in communicating the specifications to a programmer. Further, the use of pass-through SQL enables system designers, analysts and programmers to leverage their knowledge to develop mobile web applications with relative ease without any web programming. Moreover, MobiDSL allows extensibility by allowing developers to use any user-defined function in various expressions. Figure 9 presents a simplified view of MobiDSL metamoddel, which depicts various design elements/constructs at a glance.

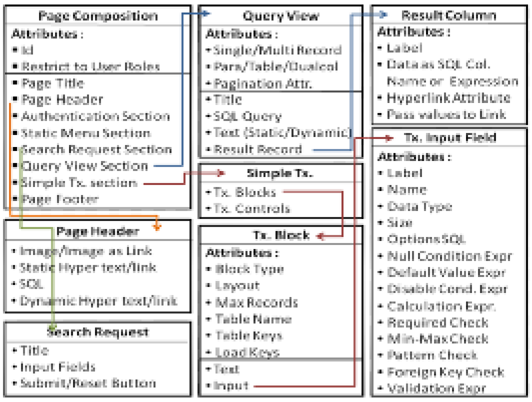


Figure 9. A simplified view of MobiDSL metamodel

4. Discussions

4.1 Application Life-Cycle Management

MobiDSL provides a reasonably sound framework for complete life-cycle management of mobile web applications:

Prototyping: As the MobiDSL specifications are at very high level almost mirroring the functional requirements, it is possible to create working prototypes using MobiDSL in similar time that might be needed to create a prototype using any prototyping tool.

Development: MobiDSL is compact DSL designed specifically for mobile web applications. The constructs provided in the DSL along with the ability to use pass through SQL and user defined functions make it reasonably adequate to cover most of the needs.

Testing: MobiDSL code, being declarative in nature, is far easier to debug than a procedural code. Moreover, the VM allows quick testing as the changes are reflected immediately in the application. This can save considerable time as the change-compile-build-deploy cycle is eliminated.

Deployment: The VM uses basic protocols (HTTP) and basic API's (CGI, pass through SQL). As a result, it can be deployed on commodity hardware or most of the existing infrastructure in an organization. It provides scalability by allowing multiple application servers, which can be added or removed without need to shut down the application. Finally, the VM does not require any special middleware typically seen in many high-end applications.

4.2 Generative vs. Interpretive Approach

We believe that a carefully crafted implementation of a VM can offer several benefits at speeds matching that of the generated code. Whereas generated code needs to be maintained, versioned, compiled and installed on an application server, these tasks are eliminated in the interpretive approach, leading to easier deployment and maintenance of the application.

5. Conclusions

Contribution. In this paper, we have looked into the question of how we can simplify the development, deployment and life cycle management of mobile web applications. The question is important because mobile web is a fast growing information delivery medium and it is vital for organizations to quickly develop systems for mobile platform with least effort. Our main contributions can be summarized as follows:

1. Identifying core requirements and design elements of mobile web applications
2. Designing a DSL for defining the complete specifications of a Page/Screen
3. Incorporating the concept of using SQL in DSL for Queries
4. Creating a VM to support the DSL

This research has resulted in development of a lightweight framework consisting of MobiDSL and its associated VM. It has been tested extensively by us, and was found to perform as per our design expectations on parameters such as coverage of problem domain, ease of development and deployment, speed of execution, scalability and reliability. This framework is also being used in industry to create mobile specific versions of certain parts of their enterprise application.

Future Work. MobiDSL being a nascent framework, is evolving continuously. Future work includes enhancing the DSL to increase functionality, providing client-side validations using JavaScript (based on device capability) and conducting a detailed comparative study of various metrics with respect to other popular web engineering methodologies.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. We also thank all those who have helped in this work.

References

[1] Mobile Web Initiative. Available: http://www.w3.org/Mobile/
[2] Mobile Web Best Practices1.0. Available: http://www.w3.org/TR/2008/REC-mobile-bp-20080729/.
[3] dotMobi Mobile Web Developer's Guide. Available: http://mobiforge.com/starting/story/dotmobi-mobile-web-developers-guide.
[4] Instant Mobilizer. Available: http://www.*instantmobilizer.com/*
[5] Janne Luoma,,Steven Kelly, Juha-Pekka Tolvanen : Defining Domain-Specific Modeling Languages: Collected Experiences. Available: http://www.metacase.com
[6] Arie van Deursen, Paul Klint, Joost Visser: Domain-Specific Languages: An Annotated Bibliography. SIGPLAN Notices 35(6): 26-36 (2000)
[7] Nunes, D. A.; Schwabe, D. : Rapid Prototyping of Web Applications Combining Domain Specific Languages and Model Driven Design. in 6th International Conference on Web Engineering (ICWE'06), ACM Press, Jul. 2006.
[8] Martin Nussbaumer, Patrick Freudenstein, Martin Gaedke: Towards DSL-based web engineering. WWW 2006: 893-894
[9] Eelco Visser: WebDSL, "A Case Study in Domain-Specific Language Engineering," in GTTSE 2007: 291-373
[10] Stefano Ceri, Piero Fraternali, Aldo Bongio: Web Modeling Language (WebML): a modeling language for designing Web sites. Computer Networks 33(1-6): 137-157 (2000)
[11] Atzmon Hen-Tov, David H. Lorenz, Lior Schachter: ModelTalk: A Framework for Developing Domain Specific Executable Models. CoRR abs/0906.3423: (2009)

MontiWeb - Model Based Development of Web Information Systems

Michael Dukaczewski Dirk Reiss
Mark Stein
Institut f. Wirtschaftsinformatik
Abt. Informationsmanagement
Technische Universität Braunschweig
http://www.tu-braunschweig.de/wi2

Bernhard Rumpe
Software Engineering
RWTH Aachen
http://www.se-rwth.de

ABSTRACT

The development process of web information systems is often tedious, error prone and usually involves redundant steps of work. Therefore, it is rather efficient to employ a model-driven approach for the systematic aspects that comprise such a system. This involves models for the data structure that shall be handled by the system (here: class diagrams), various editable and read-only presentations (views) on combinations and extractions of the underlying data (here: a special view language) and ways to connect these views and define data flow between them (here: activity diagrams).

In this paper, we present the MontiWeb approach to model and generate these aspects in a modular manner by incorperating the MontiCore framework. Therefor we shortly introduce the infrastructure that helps to develop modular systems. This involves the whole development process from defining the modeling languages to final code generation as well as all steps in between. We present the text-based class and activity diagram languages as well as a view language that are used to model our system.

1. INTRODUCTION

The development of web information systems is a domain that is rather well understood. Quite a number of web application frameworks offer means to implement such systems using a wide range of approaches in almost every modern programming language (for an overview, we refer to [30]). However, most of these frameworks still demand a vast amount of repetitive and tedious work to implement similar parts of a web application: usually a datastructure needs to be implemented following a well-defined and understood scheme, same applies to the persistence mechanisms - either manually written or by using a framework such as JPA [13]. In web systems most of the datastructure need appropriate presentations to provide CRUD (create, read, update and delete) functionality and page flow needs to be defined for each web application. Depending on the technology employed, the effort needed to implement this varies a lot: frameworks like Apache Struts [2] require the maintenance of lengthy and unreadable XML files to specify the flow between different pages.

In order to develop such a system as efficient as possible and thus to reduce laborious and error prone work of manually writing the verbose code and configuration files of a web application framework, the adoption of a model driven approach [21, 15, 14] is usually a good choice. Abstracting from implementation details, the developer can focus on specifying the essentials of the system. These are in particular (1) means to define the data structure of the application, (2) ways that enable the developer to define views on the data structure and (3) the possibility to connect these views and specify the relevant parts of a complete web application. From the models describing these aspects, one or more code generators can create many necessary parts of a web-based system. Of course the discussed languages do not cover every aspect (e.g. complicated authentication or application specific functionality is not covered), but the generators and their frameworks used provide a large part of the basic functionality.

In this paper, we present the web application modeling framework MontiWeb. One of the main targets of this approach is to come up with running prototypes early and refine those in an agile way until the final system is developed. Therefore, the MontiWeb approach does provide defaults. The discussed generators are in particular connected to target frameworks and components, that e.g. do provide persistence and a standard authentication mechanism that however can be replaced and adapted to specific needs.

Generally DSLs can be designed as graphical or text-based modeling languages. Both have its advantages and disadvantages. As we do not focus on graphical frontends, but on agile usability, we use a textual notation due to the advantages presented in [11] and the fact that both can be transformed into eachother.

The rest of the paper is organized as follows: Section 2 introduces the framework we use to implement the web application modeling languages, Section 3 describes the languages in detail, Section 4 presents related work regarding the modeling of web information systems and Section 5 concludes this paper and gives an outlook of future extensions.

2. DEVELOPING DSLS USING THE MONTICORE FRAMEWORK

As already mentioned in Section 1, we use the modeling framework MontiCore [18, 17, 19] as technological basis for MontiWeb. MontiCore is being developed at RWTH Aachen and TU Braunschweig. It allows the convenient specification of textual modeling languages and provides an extensive infrastructure to process these. It is designed for the rapid development of domain specific languages. A modeling language can be defined in an integrated format that combines both abstract and concrete syntax in one specification.

```
MontiCore-Grammar
1  grammar Classviews {
2
3    external Annotation;
4    interface ViewElement;
5
6    Classviews = Annotation* name:IDENT
7      "{" Attributes? Views* "}";
8
9    Modifier = (Editor:["editor"] |
10     Display:["display"] | Field:["field"]);
11
12   View = Annotation* Modifier name:IDENT?
13         "{" ViewElement+ "}";
14
15   ViewParameter implements ViewElement =
16     Annotation* Modifier? name:IDENT ";";
17   // ...
18 }
```

Figure 1: Definition of AST (Metamodel) and concrete textual syntax for Classviews

As shown in Figure 1, a grammar in MontiCore starts with the keyword `grammar` and is identified by a name (here: `Classviews`). Non-terminals are notated on the left hand side of a production (here: `Classviews` (6), `Modifier` (9), `View` (12) and `ViewParameter` (15)) and used on the right hand side. Keywords are enclosed in double-quotes whereas named elements have a name in front of a colon, followed by the type of element afterwards (e.g., `name` as the name and `IDENT` as the type of the predefined terminal (6, 12)). Rules can have a cardinality (e.g. `*` (6, 7) for 0 to unlimited occurence, `+` (13) for 1 to unlimited and `?` (7, 12, 16) for optional occurrence) and alternative rules (e.g. (9, 10), seperated by the pipe character (`|`)) are supported. The keyword `external` marks certain non-terminals as defined outside of the actual grammar (3) and needs to be linked to another non-terminal from a different grammar. The keyword `interface` (4) implies that the following element is a placeholder for arbitrary elements that implement this interface. Here, the non-terminal `ViewElement` can be replaced by the non-terminal `ViewParameter` or further here ommitted non-terminals (indicated by the three dots (17)).

Besides these constructs, MontiCore supports extension mechanisms such as grammar inheritance (see [19] for a more detailed description of this and the abovementioned concepts). From the grammar, several tools for model instance processing, model-to-model transformation, and code generation are generated and used within the MontiWeb tool.

3. MONTIWEB - MODELING WEB APPLICATIONS

The difficulties with developing web information systems manually were briefly described in Section 1. These problems mainly occur due to the application of different technologies that are not designed to be used together. For data persistence, a relational database management system is the common case. Modern frameworks like Struts [2] or Tapestry [3] use template engines like Velocity [28], Freemarker [9] or XML for generating the presentation. The controller is commonly written in a modern GPL like Java. Since all these technologies are developed independently but still describe the same elements on different levels, changes often need to be made in all of them. For example, if a new attribute shall be added to the data structure, all three layers are affected and need to be modified. Furthermore, often glue code in formats such as XML configuration files need to be touched as well. Thus, a model driven development approach can help a lot in these cases: convenient infrastructure provided, each of these layers can be defined in its own modeling language and describe the appropriate matter concisely. Therefore, adding one field would mainly concern one model element and reflect into all other layers automatically. Here the order in which the models are specified is not important. Modeling can be an incremental process where the different models are written in parallel and independently of eachother and then the consistency between them can be checked on the model level and be ensured through tested code generation. The three modelling languages with syntax and function in the websystem and interaction are described in the following.

3.1 Data Structure

The central aspect of a web information system is the underlying data structure. The language describing it should be flexible enough to express all necessary aspects and yet easy and domain specific enough to raise the level of abstraction above manual implementation.

Three requirements for the data structure description are set: (1) A type system (2) composability and (3) relationship between model elements. By a domain specific data type system special characteristics are assigned to the data. Thus validation of data, transformation rules, storage mechanisms and other data-specific functions are easily possible.

Composability of complex data means that one data structure can be made up from elementary data types as well as complex ones defined elsewhere in the model. The relationships between the data define mapping properties. Since class diagrams offer enough expressiveness for data modeling and are generally well-known, MontiWeb uses a textual representation of a subset of UML/P [25, 24] class diagrams to describe the data structure. In the following we explain how the chosen modeling language met the three requirements for the description of the data structure. An example of such notation is shown in Figure 2. It shows the simplified data structure of a carsharing service that consists of persons and cars. A class diagram begins with the keyword `classdiagram` and is named right after (1). It contains class definitions that are notated straight-forward with the corresonding keyword. The different attributes are defined within the class and consist of a type (e.g. `MWString` (4) which represents a domain specific implementation of a String) and a name (e.g. `name` (4)). MontiWeb distinguishes two types of classes: (a) Base classes - are similar to primitive types of Java. They do not include any attributes and are implemented in the target system according to their own rules. (b) Complex classes - contain attributes of base classes as well as other complex classes. To model relationships between two classes, associations can be used.

MontiWeb provides two types of associations. Normal associations (not shown in the example) in the generated web system are treated as link between objects, i.e. for an association between class A and B, an object of class A can be assigned to an object of class B. The second type of association is composition. It is denoted by the keyword `composition` (17-18) and the two associated classnames (`Person` and `Car`). Associations can have named roles (`keeper` and `cars`), cardinalities (`*` in this case, the ommission on the other side implies exactly 1) and directions (here, `->` which implies that a person owns cars that only exists in combination with the person). In compositions, one class is embedded into the other class, whereas the embedded object is created simultaneously with its parent object. The composition represents a part-whole or part-of relationship with a strong life cycle dependency between instances of the containing class and instances of the contained classes. This implies that if the containing class is deleted, every instance that it contains is deleted as well. Using to multiplicity and direction, other properties of the association or composition can be defined. In MontiWeb, static selection lists, such as days of the week, can be defined by enumerations (9), and can also be considered as a type of attributes. The entire data structure is distributed over several class diagrams. A class diagram is an excerpt of the overall system. The source code in Figure 2 shows a part of the car sharing web system.

```
Classdiagram
1  classdiagram Carsharing {
2
3    class Person {
4      MWString name;
5      Email email;
6      Number age;
7    }
8
9    enum Brand {AUDI, BMW, VW;}
10
11   class Car {
12     Brand brand;
13     Number numSeats;
14     MWDate constYear;
15   }
16
17   composition Person (keeper)
18      -> (cars) Car [*];
19 }
```

Figure 2: Datastructure of a Carsharing application

3.2 View Structure

The presentation layer is responsible for rendering the data and providing the interface between a human user and the web information system. Since the main focus of MontiWeb is the domain of data-intensive web applications, the modeling language used offers means to conveniently specify data entry and presentation rather than extensive structures to detailly describe pretty interfaces. Nevertheless, the generated layout can be altered by the common means of adjusting the templates for code generation and the inclusion of Cascading Style Sheets (CSS) and thus fitted to a certain (corporate) design. From a language to specify views of a web system, we demand the following: (a) different, possibly limited views on the underlying data structure must be specifiable, (b) views are composable, i.e. once defined views can be composed to and reused in other ones, (c) static parts (e.g. text or images) can be included in dynamic views on the data and (d) web specific convenience functionality like validation, filtering, sorting data etc. can be modeled with the provided language. Since the UML does not offer any way to specify such features, we developed a domain specific Classview language which allows the specification of different views on a certain class from a class diagram. Each Classview file includes named views on exactly one class (and thus fulfilling the abovementioned requirement (a)). An example of such for class `Person` is depicted in Figure 3.

```
Classviews
1  Person {
2
3    attributes {
4      @Required
5      @Length(min=3, max=30)
6      name;
7      @Required
8      age;
9    }
10
11   display protectedMail {
12     name;
13     @AsImage(alt=false)
14     email;
15     cars;
16   }
17
18   display welcome {
19     text {Welcome to Carsharing Service}
20     include protectedMail;
21     age;
22   }
23
24   @Captcha
25   editor registration {
26     name;
27     email;
28     age;
29     cars;
30   }
31
32   display error {
33     @Warning
34     text {You are not old enough!}
35   }
36 }
```

Figure 3: Example of Classviews

Within MontiWeb, special functionality (such as the ones noted above in (d)) is encoded in a syntax that is borrowed from Java annotations. These begin with an ampersand (`@`) and may have additional attribute-value pairs in parens appended to it (e.g. (5)). For MontiWeb, we already offer a rich selection of predefined domain-specific annotations - some of them shown in the example and explained in the following. The rules within the element `attributes` (3-9) apply to all views within the classview file. Here these imply that the attributes `name` and `age` are obligatory to enter (`@Required` (4, 7)) and `name` may appear 3 to 30 chars (`@Length(min=3, max=30)` (5). These result in the generation of according AJAX verifictaion mechanisms. Subsequently, the different views are specified. These begin with the type of view (here: `display` (11, 18, 32) for views that simply output the data and `editor` (25) that renders the appropriate input fields for the classes' attributes) and are followed by a name. The view `protectedMail` renders the name, email address and cars data of a person whereas the email address is being transformed to an image (caused by the web-specific annotation `@AsImage` to avoid automatic email address harvesting). The `welcome` view displays some static text (19), does furthermore include the `protectedMail` view and displays a persons age. This functionality satisfies the demands (b) and (c) from above. The `registration` view is an editor view and thus provides input fields for `name`, `email` and `age` of a person and – as `cars` denotes the composition of car objects within a person – means to associate such objects to a person. The annotation `@Captcha` (24) produces a captcha field on this view. Finally, the view `error` (32-35) simply consists of a static text that is rendered in a manner that indicates a warning.

An example of how the `registration` view could be rendered is shown in Figure 4.

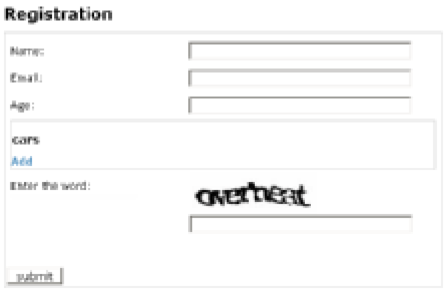


Figure 4: View "editor"

3.3 Control- and Dataflow

Defining only the data structure and different views on it suffices for generating basic web information systems that allow rudimentary data manipulation functionality like entering and saving, showing and updating the data. To create more complex web applications, we need means to model both, control and data flow between the different pages or views respectively. For this purpose, we use a profile of UML activity diagrams [22] in textual notation. An example of an activity diagram is shown in Figure 5. It describes a process of user registration where a user enters his user data and is then directed to either a welcome page (in case his age is greater than 18) or an error page (if the age is smaller than 18).

An activity diagram starts with the keyword `activity` followed by the activities' name (here: `UserRegistration`). Actions (introduced by the keyword `action` (3, 8, 13)) posses

Activity Diagram

```
 1 activity UserRegistration {
 2
 3   action Registration {
 4     out: Person p;
 5     view : p = Person.registration();
 6   }
 7
 8   action Welcome {
 9     in: Person p;
10     view : Person.welcome(p);
11   }
12
13   action Error {
14     in: Person p;
15     view : Person.registrationError(p);
16   }
17
18   initial -> Registration;
19   Registration.p -> [p.age >= 18] Welcome.p
20                   | [p.age < 18] Error.p;
21   Welcome | Error -> final;
22 }
```

Figure 5: Example of Activity Diagrams

a name as well and include different contents: `in` (9, 14) and `out` (4) followed by an attribute type (`Person`) and attribute name (`p`) specify input and output parameters of an action. The keyword `view` (5, 10, 15) indicates the kind of content of an action. The view itself is referenced by its name and either can take an object as argument (10, 15) to initialize the view or return an object which is assigned to an output parameter (5). Transitions within an activity are represented by an arrow symbol (`->` (18, 19, 21)) and may contain several sources and targets. The keywords `initial` and `final` denote start and final nodes of an activity and the pipe character (`|` (20, 21)) depicts alternative flows - with conditions on the right hand side (19, 20) or as alternative routes to the final node (21). Object flow is modeled by appending the parameter name to the action name and for simple control flow, these parameters are left out.

Besides these notation elements, concepts such as parallel flow, hierarchical actions (which themselves are specified by an activity) and roles to which actions can be assigned are supported as well but omitted in this paper for the sake of space. Furthermore, different content can be included in an action. Presently, the inclusion of Java code is supported along the already mentioned view calls.

3.4 Aggregation (Interaction) of Component Specific Languages

The described models define three views on a whole system. They are developed and specified independently from each other to maintain clean seperation of the different components. Nevertheless the model parts have some well-defined connection points. Elements that are defined in one model are referenced from another (e.g., views are referenced from an action). The inter-model-relationships are essential for completeness and correctness of the whole system and finally define its behavior.

When developing modeling languages from scratch for parts of a domain, first and foremost only these parts are considered. However, although they will work in isolation, they are often used in combination to model the complete system. Therefore, the notation of a language must provide means to connect to other components.

Interaction between modeling languages can be realized with different mechanisms [26], e.g. by embedding one language into another like SQL is embedded into a GPL like Java. In MontiWeb, the inter-language interaction is realized by using the pipeline pattern [26]. There, the different languages are independent but still implicitly connected, i.e. the implicit relationships are explicitly checked within the generation process. The visibility between the MontiWeb models is depicted in Figure 6. The controller functionality is realized similar to the Application Controller pattern [8]. Here, class diagrams are completely independent from the rest of the model. Neither the data presentation (classviews) nor the flow control (activity diagrams) are of importance for the data definition and thus can not be referenced from there. Classviews depend on the data structure as they define explicit views thereof and contain references (e.g. to classnames, attribute names and types or association names) to it. Classviews do not reference activity diagrams, vice versa activity diagrams reference classviews by name. As the control flow defines the central logic of a web information system, both class diagrams and classviews are referenced from there. To maintain consistency between these models, inter-model checks are performed through, e.g. modular symbol tables. Thus the existence of a referenced view or class can be verified.

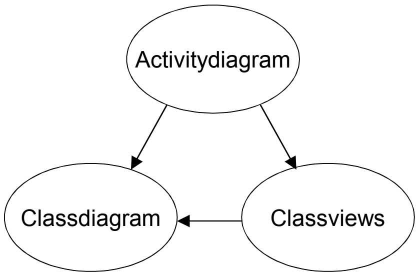


Figure 6: Models of MontiWeb and their dependencies

4. RELATED WORK

Similar approaches of modeling of web information systems can be classified into (a) modeling using graphical languages, and (b) modeling using textual languages.

One graphical modeling tool in the domain of web information systems is WebML. Unlike MontiWeb, WebML distinguishes two domain segments: (a) data design concerns the specification of data structures and (b) hypertext design is used to describe the structure of web pages [6]. Both of these languages incorporate UML class diagrams. For hypertext design, predefined classes like Entry for the generation of a web form or Data to display a class are used. The navigation structure is depicted by directed associations between classes. Furthermore, WebML supports a XML based textual modeling language which lacks tool support. Therefore, the use of their own graphical modeling tool is favored [5].

UWE (UML-based Web Engineering) [16] follows a similar approach as WebML. It uses class diagrams for data structure specification and, like MontiWeb, uses acivity diagrams to describe the modeling of workflow. UWE's notation is a graphical one as well. Like WebML, the UWE models can be exported in an XML format.

Another tool that uses graphical modeling is AndroMDA [1]. AndroMDA does not have its own editor yet, but uses XMI as input format which is supported by some UML Editors. Like MontiWeb, it uses class diagrams for data structure and activity diagrams for workflow description. AndroMDA does not offer a specific language to describe the view aspect of a web information system, but generates it from extra class diagrams that have to be specified additionally. To get a working application, all parts have to be provided. A generation of standard behavior as MontiWeb does is not supported.

As a textual modeling approach, WebDSL [29] follows a similar approach as MontiWeb. The language there is specified using SDF [12] and Stratego/XT [4] for language transformation. They use a purely domain specific modeling language and is not leaned on UML.

The Taylor project [27] follows an MDA approach to model and develop JEE applications. The models are created using Eclipse Graphical Modeling Framework (GMF) [10] and are stored in XMI format by incorporating EclipseUML [7]. As notation for data structure, Taylor uses class diagrams, business processes are defined by activity diagrams as well. The navigation structure between pages is specified by a state machine language where states depict pages and transitions links from page to page.

Another popular approach for generating web information systems is Ruby on Rails [23]. Although it is not a pure model based approach, a prototype application can be generated using the Ruby on Rails scaffold mechanism. From a simple model in a Rails-specific notation and a HTML-based view template language, CRUD functionality and a very basic controller can be generated. However, unlike MontiWeb, the focus of Rails is the manual programming of all three components, aided by extensive web-specific functionality provided by the language.

The Mod4j (Modeling for Java) [20] project aims at the efficient development of administrative enterprise applications by employing a model driven approach. Like MontiWeb, Mod4j seperates the application into its different aspects and offers a modeling language for each. The Business Domain Model is represented by an UML class diagram. Page flow is modeled using a specific Service Model and the presentation in the application has its own modeling language as well. Mod4j is based on Eclipse technology and uses XText [31] for the development of languages.

5. CONCLUSION AND FUTURE WORK

In this paper we described our approach to model and generate web information systems to tackle the insufficiences that occur when developing such systems manually. Especially the difficulties caused by the combination of normally orthogonal frameworks are approached. Within MontiWeb, we use three languages for the three main segments of a web information system. Two of them come from the UML/P, one language (classviews) is completely new defined. These languages reflect the requirements of each domain component and were adapted to their specific needs.

The currently reached status involves pretty stable languages, as discussed here, and appropriate generation tools. Furthermore a number of presentation forms for various data types (such as Date, String etc. are defined). We currently work on extensions of the provided functionality in various ways. This includes e.g. components for more fine grained security, identification and authentication as well as the possibility to easily integrate predefined (third-party) components that provide application specific functionality. We plan to further extend and complete the already used languages (e.g. include inheritance in class diagrams) and incorporate new ones to model not yet covered aspects of a web information system (such as use case diagrams for requirements modeling). Furthermore, we think of generation of a modular API to access the generated system via SOA-services or add SOA-functionality provided by other servers.

6. REFERENCES

[1] AndroMDA Homepage `http://www.andromda.org/`.

[2] Apache Struts Homepage `http://struts.apache.org/`.

[3] Apache Tapestry Homepage `http://tapestry.apache.org/`.

[4] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, 72:52–70, 2008.

[5] S. Ceri, P. Fraternali, and A. Bongio. Web Modeling Language (WebML): a modeling language for designing Web sites. *Computer Networks*, 33(1-6):137–157, June 2000.

[6] S. Ceri, P. Fraternali, and M. Matera. Conceptual modeling of data-intensive web applications. *IEEE Internet Computing*, 6(4):20–30, 2002.

[7] Eclipse UML Project `http://www.eclipse.org/modeling/mdt/?project=uml2`.

[8] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[9] Freemarker Website `http://freemarker.org/`.

[10] Graphical Modeling Framework Website. `http://www.eclipse.org/gmf/`.

[11] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering*, 2007.

[12] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - Reference Manual. *Sigplan Notices*, 24(11):43–75, 1989.

[13] Java Persistence API `http://java.sun.com/javaee/-technologies/persistence.jsp`.

[14] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008.

[15] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[16] N. Koch and A. Kraus. The expressive power of UML-based engineering. In *Second International Workshop on Web Oriented Software Technology (CYTED)*, 2002.

[17] H. Krahn, B. Rumpe, and S. Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling 2007*, 2007.

[18] H. Krahn, B. Rumpe, and S. Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Proceedings of Models 2007*, pages 286–300, 2007.

[19] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In *Proceedings of Tools Europe*, 2008.

[20] Mod4j Homepage `http://www.mod4j.org/`.

[21] Object Management Group. MDA Guide Version 1.0.1 (2003-06-01), June 2003. http://www.omg.org/docs/omg/03-06-01.pdf.

[22] Object Management Group. Unified Modeling Language: Superstructure Version 2.1.2 (07-11-02), 2007. http://www.omg.org/docs/formal/07-11-02.pdf.

[23] Ruby on Rails Website `http://rubyonrails.org`.

[24] B. Rumpe. *Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring*. Springer, 2004.

[25] B. Rumpe. *Modellierung mit UML*. Springer, 2004.

[26] D. Spinellis. Notable Design Patterns for Domain Specific Languages. *Journal of Systems and Software*, 56(1):91–99, Feb. 2001.

[27] Taylor Homepage (http://taylor.sourceforge.net/).

[28] Velocity Website `http://velocity.apache.org/`.

[29] E. Visser. WebDSL: A Case Study in Domain-Specific Language Engineering. Technical Report TUD-SERG-2008-023, Delft University of Technology, Software Engineering Research Group, 2008.

[30] I. Vosloo and D. G. Kourie. Server-Centric Web Frameworks: An Overview. *ACM Computing Surveys*, 40(2):1–33, 2008.

[31] Xtext Homepage `http://www.eclipse.org/Xtext/`.

ProcDSL + ProcEd - a Web-based Editing Solution for Domain Specific Process-Engineering

Christian Berger Tim Gülke Bernhard Rumpe

RWTH Aachen University
Software Engineering Group
Ahornstraße 55
52074 Aachen, Germany

www.se-rwth.de

ABSTRACT

In a high-tech country products are becoming rapidly more complex. To manage the development process as well as to encounter unforeseen challenges, the understanding and thus the explicit modeling of organizational workflows is more important than ever. However, available tools to support this work, in most cases force a new notation upon the company or cannot be adapted to a given publication layout in a reasonable amount of time. Additionally, collaboration among colleagues as well as different business units is complicated and less supported. Since it is of vital importance for a company to be able to change its processes fast and adapt itself to new market situations, the need for tools supporting this evolution is equally crucial. In this paper we present a domain specific language (DSL) developed for modeling a company's workflows. Furthermore, the DSL is embedded in a web-based editor providing transparent access using modern web 2.0 technologies. Results of the DSL's as well as the editor's application to document, model, and improve selected workflows of a German automotive manufacturer are presented.

1. INTRODUCTION AND MOTIVATION

In today's world of business-processes, modeling becomes a vital factor in an organization's change and process management and even daily routines. This in particular holds for the development of complex machines, such as airplanes, cars or trains. Each of these domains has their own specific problems, e.g. induced through supplier integration, need for quality certification, development for individual customers or the mass market, etc.

It is therefore not surprising that there is no unique solution for the management of these processes. Therefore, it is only natural to find company-specific layouts of process-descriptions and publications in almost every firm. Unfortunately, tools meant to support organizations in planning and developing their processes, often force their own layout, notation and logic upon their users. Although this might be considered easier and even 'better' than what the company is used to, we found it is one main reason to see printed Microsoft PowerPoint slides and similar documents to arise all over office walls. Big organizations need a certain amount of time to agree on a specific appearance of their process-documents and even longer to publicize this throughout the company. And even worse, the meaning of icons or the position of images tends to change during a company's evolution. Programs like [6] or [2] are not able to be easily adapted to appear like what people know and have worked with already. This fact clearly shows the need for a modular tool which can be adapted with considerably less effort than any others available.

Furthermore, many currently available tools are single user applications with only limited possibility for company-wide collaboration. Using MontiCore [5], a framework for developing domain specific languages, we developed a web-based editor for modeling organizational workflows that uses a DSL's instances as input and output. This DSL was developed together with company-experts to ensure correctness and completeness. The editor's interface then was constructed separately, so there was a clean cut between the logic and its representation. This enables us to change either the logic behind a process-plan or the frontends' appearance without touching the other. Process modeling is then being performed by the end user through a web-browser to gain the amount of flexibility necessary in today's quickly changing world. AJAX technology enables us to construct an interface almost as powerful as a traditional application's one.

This paper is structured as follows. First, a brief overview of MontiCore is presented. Following, the design considerations and the implementation of a DSL to model organizational workflows are discussed. This DSL is embedded in a web-based process editor which is presented afterwards. Finally, the DSL as well as the editor's application is shown on an example from the automotive domain.

2. MONTICORE – A FRAMEWORK FOR DEVELOPING DOMAIN SPECIFIC LANGUAGES

MontiCore is a framework for developing textual domain specific languages from the Department of Software Engineering at RWTH Aachen University. It supports grammar-based language design as well as meta-modeling concepts by providing one language for abstract and concrete syntax definition. Using a given definition it generates automatically a lexer, a parser, and classes for an abstract syntax graph (ASG) describing the language's structure. At runtime, these classes represent a successfully parsed and processed instance of a given language [5, 7, 8, 9].

Generated artifacts and MontiCore itself are coded in Java. Due to its sophisticated language processing concepts and its support for Java which is also used by the technology we intended to use for realizing the web-based editor, we have chosen MontiCore for defining the language and for processing instances at runtime.

3. PROCDSL – A DSL FOR PROCESS DESCRIPTIONS

We propose a domain specific language to represent the company's organizational workflow processing. The reason we used a DSL to formalize the logic behind a process-plan was the complex structure of those plans, hidden behind a rather simple appearance. Basically, a milestone's appearance in a specific plan was determined by the organizational view, consisting of a layer and unit combination, the plan represented. One unit might be only participating to the milestone's result while another one is responsible for it. Both types of access are represented through different icons in their unit's process-plan.

Without the use of MontiCore and a DSL, we would only have been able to construct an application that suits the current needs and requirements as we understood them. In case of a sudden change of the appearance or logic of those process-plans, the application would have to be reconstructed in a time-consuming way. Through our separation it is now possible to change either the model or the graphical representation without touching the other. The DSL, which we modeled together with chosen experts, enabled us to already start the development of the editor-frontend while still being in the process of figuring out the logic's details behind the plans. This will also save resources later if for example different views will be needed for the same data. We predict that in near future, a plan's layout will change again or a new organizational layer might be implemented - in that case the model or the editor can be changed quickly without the need to rewrite a whole database scheme and an applications access to it.

The main advantage over pure visual process modeling tools is its formal specification. Additionally, a textually defined DSL can be simply embedded in different contexts and therefore easily reused. In the following, we present briefly the DSL we designed for modeling organizational workflows considering the following design criteria.

- *Intuitional Representation.* Instead of using XML for defining workflows we chose a much more simple representation to avoid XML's verbosity and redundancy in its data description. Thus, a better readability for the user can be achieved if DSL's instances are processed without a graphical editor.

- *Small Data Format.* Since the language is intended to be used in a web-based context, large entities of organizational workflow descriptions would cause lots of bandwidth consumption. Thus, a small data format to be exchanged with a server is desirable.

- *Reusability.* The language itself is primarily intended to be used with a graphical web-based editor to support process engineers. However, having a formally defined and application-independent process description, language's instances can be easily exchanged among the same application. Moreover, other tools can be used for checking semantic constraints on the one hand or to transform an instance into another data format on the other hand.

- *Versioning.* Regardless if an available solution like Subversion is used or a domain-specific (e.g. graphical) one is programmed it is obvious that textual formats are easier to put under version control as well to track and compare changes.

To ensure usefulness, domain experts from the company were heavily involved in the development of the DSL. Using a simple UML-representation of the DLS's structure, we were able to communicate in a productive way.

In Fig. 1, an excerpt of our grammar is shown. Technically, MontiCore accepts productions with EBNF-like right hand sides. Nonterminals (like `Milestone` or `String`) can be preceded by attribute names (like in `name:String`). Attribute names can also be attached to terminals like `"Scope"` or `"resp"` denoting, whether the keyword was detected.

Lines 1-5 contain the grammar's start symbol. The workflow description starts with a header containing some meta-information about the current instance followed by a list of milestones. Every milestone has a name, a description, and several other properties of which some are included in 1, lines 9-13. Line 11 positions the milestone relatively to a timeline. As already mentioned, the need to separate the logic behind a process-plan and it its actual graphical representation was crucial. Therefore, during development of the DSL, we made sure not to mix graphical information like icon positions, colors, and the like with logic-related things. As a result, an instance of the given DSL does not only represent a milestone-plan like the one shown in Fig.4 which was used as a blueprint for the DSL, it also enables developers to get different graphical representations out of it (e.g. simple lists of milestones, a specific view on inputs and outputs of a milestone or the involvement of a layer in process activities).

Besides an informal description, a milestone has a concrete result which can be any appropriate artifact depending on a specific workflow. Different scopes and layers can access

```
MontiCore-Grammar
 1 ProcessFile =
 2   "process"
 3   ProcessHeader
 4   :Milestone*
 5   Process
 6   :Scope*
 7   "end";
 8 ...
 9 Milestone = "milestone" Name
10 ...
11  "position" TimelinePosition:Number
12  "result" Result:Result*
13  "description" Description:String;
14 ...
15 Scope = "scope" Name
16  "description" Description:String
17  r:Responsibility*;
18
19 Responsibility = "responsibility"
20  (responsible:["resp"]
21  | contributing:["cont"]
22  | noticing:["noti"])
23  "asmilestone" asMilestone:STRING;
24 ...
25 associations {
26   Responsibility.milestone * -> 1 Milestone;
27 }
28
29 concept sreference {
30  ResponsibilityMilestone:
31    Responsibility.asMilestone = Milestone.name;
32 }
```

Figure 1: Excerpt of our grammar to describe organizational workflows.

a milestone in different ways, like being responsible or just contributing to the result. In this case, a scope is a specific organizational unit within a layer, like *manufacturing* within the layer *departments*. Combined, these selections define different views on the whole set of milestone-data. A scope's responsibilities are described in lines 19-23. Every scope is either *directly responsible* for fulfilling a sub-process associated with this milestone, *contributing* for a concrete milestone or only *noticing* the state of a sub-process.

Using the concept of automatically set associations provided by MontiCore in line 25-32, the responsibilities' milestones are navigably associated with an ASG node describing a milestone. The following lines starting at line 27 describe the way a milestone is mapped by its (unique) name to the corresponding responsibility-object's association.

For validating given values of concrete DSL's instances object which traverses the ASG, generated by MontiCore can be defined. For example, a time-validating visitor can be used to check the semantic constraints whether the start time of a given milestone is prior to its end time regarding to the underlying timeline specification, which can be either a regular calendar or a simple sequence of weeks.

Using the grammar outlined in this section, we designed and implemented a graphical web-based editor which is described in the following.

4. GRAPHICAL EDITOR FOR PROCDSL USING WEB 2.0 TECHNOLOGIES

We wanted the graphical editor to be as easily usable as possible combined with the flexibility a web-application gives us regarding deployment and maintenance. AJAX enables developers to design web-applications that make use of asynchronous callbacks rather than of synchronized ones. Therefore, the traditional request-response-paradigm is no longer the limiting factor in a web-application's interface. Using AJAX different parts of the website can be loaded dynamically providing a great range of possibilities to the developer to design the application. For more information about the AJAX technology see [12].

Since the overall layout was already fixed due to the fact that we were working with a company which had already specified its appearance for process descriptions, it was clear that the editor should not be a generic canvas, but an aid to work in that given layout. However, it should use the formalism provided with the DSL to keep users from inventing new icons and limit them to correct instances.

We selected the Google Web Toolkit[4] as our main framework which enabled us to write Java-code instead of JavaScript for the web-interface. Through this, a highly interactive web-based application combined with proper testing and a decent coding-style was possible with much less effort than a traditional one would have required. The implementation of Drag&Drop-capabilities as displayed in Fig.2 for canvas-objects as well as dialog-windows is another factor that makes the interface a lot more comfortable for users. Through asynchronous callbacks, drafts can be saved and restored automatically.

As one can see in Fig.3, the main window is divided in three different areas. The largest is used by the actual milestone-plan, while the other two keep a toolbox to drag objects out from onto the plan and an object-inspector. The latter allows users to look into a chosen item's details. To select an object on the plan, it can simply be clicked on.

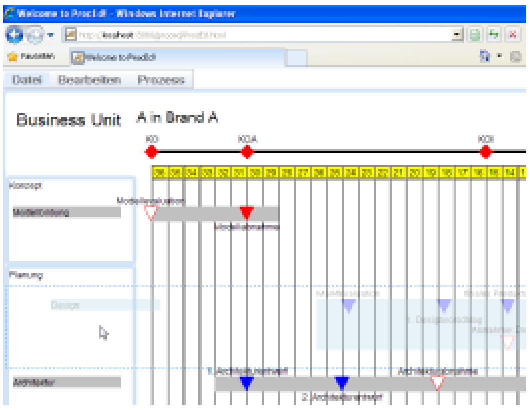


Figure 2: Web-based dragging and dropping of items and collections of items.

As input and output, an instance of the above defined grammar is used. After upload, the ASG is constructed from the file through the MontiCore-generated tools, although these objects are kept separate from the ones behind the displayed items. This separation enables us to replace both sides, grammar-generated objects and data-objects, without changing much at the corresponding one in case an engineering- or design-related update is necessary. No instance of an ASG-object is kept in a visualizable one and vice versa to achieve a very clean separation between the two worlds. Since the editor does not get any display-related information from the grammar, it has to decide itself on the positioning and use of visual elements such as icons, colors, etc.

To keep everything synchronized and to reduce computational effort and bandwidth, a central class containing a hashmap keeps all objects and links them to an icon-file that the user will finally see. This pattern makes searching and working in general with the data easier as if ASG-objects would keep their visualized counterparts themselves instead. A Command-pattern makes sure user-input is handled correctly and distributed to the right object and also adds an undo-/redo-functionality to the editor.

The web-application itself is secured through an SSL-connection and a required login provides a user-environment that lets a user keep a list of files he is working on. This part of the application could be extended, for example with functions like shared comments.

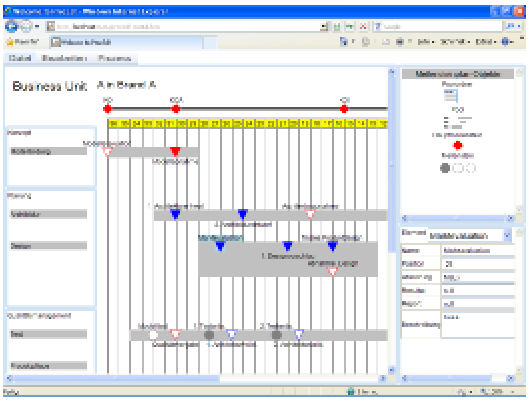


Figure 3: Screenshot of ProcEd, a web-based editor for instances of ProcDSLs.

5. APPLICATION AT AN AUTOMOTIVE MANUFACTURER

The already fixed layout of graphical representations of the corporate processes made questions about the application's appearance simple. The requirements gathered for this yielded to a login-screen, traditional file-menus, etc. Example process files in Microsoft Power Point like the one shown in Fig.4 specified the canvas' layout. The real difficulties therefore lay in the business-objects model and the different influences the classes have on each other.

We used a graphical representation of our model to be able to discuss it with selected domain-experts who had no education in computer science. UML was a good choice due to easily understandable class diagrams. However, we needed several iterations to get to a complete model-specification.

The clearly separated parts in the software though made it easy for us to keep up an agile workflow. While we could implement more and more of the interface, the model itself could be improved independently only requiring to generate a new lexer and parser using MontiCore to process the modified version, and correct a function call or the like in the separated part. Compared to a traditional approach which would have forced us to complete the model first and then build the application depending on it; using the aforementioned approach we used during development we simply could not only integrate but also embrace changes desired by the customer [1].

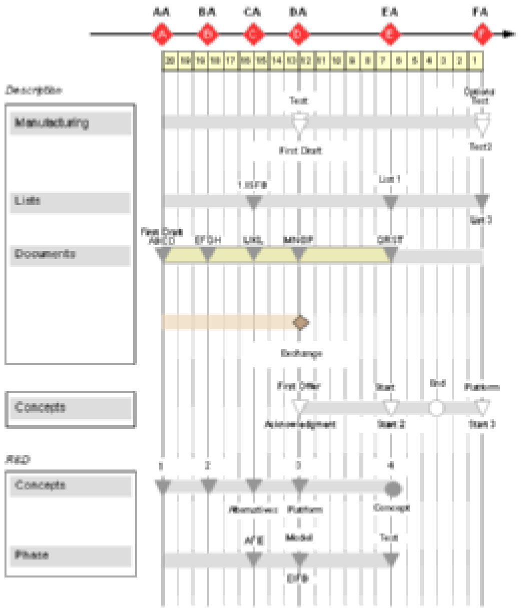


Figure 4: Elements for process description at a German automotive manufacturer.

6. RELATED WORK

The usage of DSLs in web-applications has received increased interest in the last years. But as outlined in [10] or [11], these activities did only focus on modeling an application's architecture and related workflows.

Our approach is different, because we did not use the DSL to define workflows, but to get a data exchange format that also serves as business layer in the resulting application. A change in the DSL would not result in a whole new application layout, only in differently working interfaces, leaving the former with the customer agreed on GUI intact. This is crucial, since the organizational layout for process documents is already company-wide communicated and approved. To understand the meaning behind different layers and associations though can be a tough job which needs flexible tools that will only change parts of the code that need to be changed with minimum effort.

The choice of Google's GWT as the framework used for realizing the web-editor was based on the excellent Eclipse integration and the number of features it includes. However, the most important factor, compared for example to [3], was the fact that GWT enables the developer to work in plain Java without having to care about data exchange or even JavaScript on the user's end. This is clearly an advantage because it decreases development time and simplifies testing and source code documentation. Moreover, as already discussed earlier, MontiCore generates the classes representing the grammar's ASG in Java which could be easily integrated with GWT.

Reasons a new DSL was used instead of implementing one of the business modeling languages available were the very company-specific process-layout on the one hand and the missing or incomplete formal specification of those languages on the other hand. As for example [13] notice, the Business Process Modeling Language (BPMN) lacks several concepts, like sub-processes with more than one instance, is partially ambiguous in its definition and has an incomplete mapping to the formal representation WS-Business Process Execution Language (BEPL). Moreover, BPMN did not let us represent the company-specifics we needed to be able to model, like a milestone's different meanings defined by the way different layers access it. This was a crucial fact since we needed to be able to display different views from different layers of the company onto the same sets of milestones and the connections between them. If one milestone changes, it has to be updated in every representation. This can only be achieved with a data-model representing exactly the company's structure.

7. CONCLUSION

In this paper, a formal, textual-based domain specific language for defining workflows was presented. Using this language, both documentation and modeling of organizational processes of a company is supported and also given instances of the DSL representing several workflows can be inspected.

For supporting a process engineer in modeling, documenting, and integrating different workflows, a graphical web-based editor using modern web 2.0 technologies was provided. Using this editor, workflows can be transparently presented and updated nearly everywhere in a company using a web-browser with state-of-the-art technologies already built-in.

The main contributions of this work – the formal description of organizational workflows on the one hand, and transparent access to the DSL's instances nearly everywhere on the other hand – provide valuable support for a process engineer's daily work. Furthermore, formal and machine-processable analysis of the DSL's instances can be realized both to check currently implemented workflows and to simulate changes in a company's processes to perform what-if-analysis.

With the MontiCore framework and toolkit, it was easy and efficient to define and implement the DSL-part of the editor, including a lexer, a parser, ASG classes, and standard context conditions. This and other examples from different domains have shown that the MontiCore infrastructure provides efficient techniques to develop DSL-based tools.

8. REFERENCES

[1] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for the Agile Software Development, 2001.

[2] BOC ADONIS `http://www.boc-group.com/`.

[3] Echo3 `http://echo.nextapp.com/site/echo3`.

[4] Google Web Toolkit `http://code.google.com/intl/de/webtoolkit/`.

[5] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[6] IDS Scheer ARIS Business Designer `http://www.ids-scheer.de/de/ARIS_ARIS_Platform/7796.html`.

[7] H. Krahn, B. Rumpe, and S. Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling 2007*, 2007.

[8] H. Krahn, B. Rumpe, and S. Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Proceedings of Models 2007*, 2007.

[9] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In *Proceedings of Tools Europe*, 2008.

[10] M. Nussbaumer, P. Freudenstein, and M. Gaedke. The impact of DSLs for assembling web applications. *Engineering Letters*, 13(3):387–396, 2006.

[11] M. Nussbaumer, P. Freudenstein, and M. Gaedke. Towards DSL-based web engineering. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 893–894, New York, NY, USA, 2006. ACM.

[12] L. D. Paulson. Building Rich Web Applications with Ajax. *Computer*, 38(10):14–17, 2005.

[13] P. Wohed, W. M. van der Aalst, M. Dumas, A. H. ter Hofstede, and N. Russell. *Business Process Management*, chapter On the Suitability of BPMN for Business Process Modelling, pages 161–176. Springer Berlin, 2006.

Model-View-Controller Architecture Specific Model Transformation

Hiroshi Kazato
Tokyo Institute of Technology /
NTT DATA CORPORATION
Tokyo 152–8552, Japan
kazato@se.cs.titech.ac.jp

Rafael Weiß
Tokyo Institute of Technology
Tokyo 152–8552, Japan
rweiss@se.cs.titech.ac.jp

Shinpei Hayashi
Tokyo Institute of Technology
Tokyo 152–8552, Japan
hayashi@se.cs.titech.ac.jp

Takashi Kobayashi
Nagoya University
Nagoya 464–8601, Japan
tkobaya@is.nagoya-u.ac.jp

Motoshi Saeki
Tokyo Institute of Technology
Tokyo 152–8552, Japan
saeki@se.cs.titech.ac.jp

ABSTRACT

In this paper, we propose a model-driven development technique specific to the *Model-View-Controller* architecture domain. Even though a lot of application frameworks and source code generators are available for implementing this architecture, they do depend on implementation specific concepts, which take much effort to learn and use them. To address this issue, we define a UML profile to capture architectural concepts directly in a model and provide a bunch of transformation mappings for each supported platform, in order to bridge between architectural and implementation concepts. By applying these model transformations together with source code generators, our MVC-based model can be mapped to various kind of platforms. Since we restrict a domain into MVC architecture only, automating model transformation to source code is possible. We have prototyped a supporting tool and evaluated feasibility of our approach through a case study. It demonstrates model transformations specific to MVC architecture can produce source code for two different platforms.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures, Patterns*; D.4.7 [**Operating Systems**]: Organization and Design—*Interactive systems*

General Terms

Design, Languages

Keywords

Model-Driven Development, Model Transformation, Model-View-Controller Architecture, UML Profile

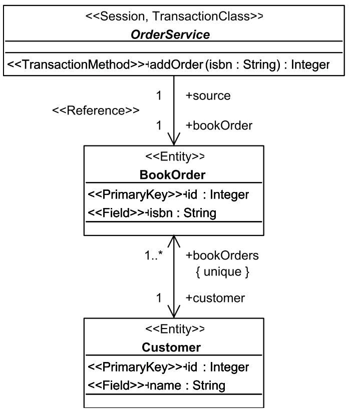
1. INTRODUCTION

Model-driven development (MDD) [16] is a development paradigm, which places models as primary artifacts and derives executable software by means of model transformation. It aims to increase productivity, maintainability and reusability of models by raising the level of abstraction above general-purpose programming and modeling languages. Some MDD tools, such as openArchitectureWare (oAW) [18] and AndroMDA [1], use their own UML profiles to include their necessary information into UML models. Since this kind of tools transform profiled UML models into source code, hereafter we refer to them as (model-driven) code generators. Along with the recent evolution in model transformation techniques, they have shown the possibility and effectiveness of MDD in practice to some extent.

However, because of the diversity of implementation platforms and code generators, there are a lot of UML profiles corresponding to various implementation concepts, and thus it is a labor-intensive and error-prone task to build, maintain and reuse these models. To cope with these problems, we should think of another level of abstraction by identifying similarities of various kind of implementation platforms and using code generators as building blocks.

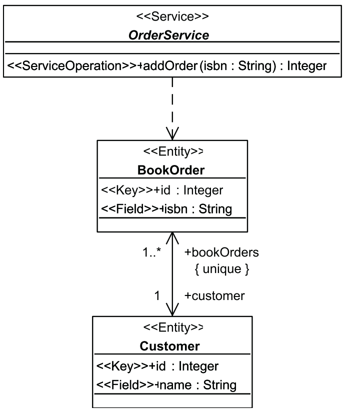
In this paper, we propose a model-driven approach called ACCURATE, in which the *Model-View-Controller* architecture style is used to capture design concepts in a user-interactive application as well as to classify implementation platforms such as application frameworks and libraries. More specifically, we define a UML profile to describe architectural concepts directly in a model. Using this profile as a pivot [3], a bunch of transformation mappings is provided for each supported platform, in order to bridge between architectural and implementation concepts. By applying these mappings and code generators in sequence, our MVC-based model can be transformed into implementation models and source code for various platforms. Automating these transformations is feasible because we only cover a restricted architecture domain. The main contribution of this paper is to propose a model-driven approach specific to the *Model-View-Controller* architecture.

The rest of this paper is organized as follows. In the next section we explain our motivation by a brief example. Section 3 presents the ACCURATE approach. Section 4 briefly introduces the prototype implementation of our toolkit and following Sect. 5 evaluates the approach through a case study of an address book application. In Sect. 6, we survey some related works and close with conclusion and future work in Sect. 7.



<<Session, TransactionClass>>
OrderService
<<TransactionMethod>>+addOrder (isbn : String) : Integer
1 +source
<<Reference>>
1 +bookOrder
<<Entity>>
BookOrder
<<PrimaryKey>>+id : Integer
<<Field>>+isbn : String
1..* +bookOrders { unique }
1 +customer
<<Entity>>
Customer
<<PrimaryKey>>+id : Integer
<<Field>>+name : String

(a) EJB Profile



(b) Spring and Hibernate Profiles

Figure 1: PSM Examples Using Profiles for the Fornax-Platform [10]

2. MOTIVATING EXAMPLE

Class diagrams shown in Fig. 1 are examples taken from two different platform specific models (PSMs), one uses a profile for EJB and the other does a combination of profiles for the Spring Framework and Hibernate. Both of them specify almost same functionality, i.e. object/relational mapping between Java and relational databases, but they are different from a technical view because of their platform-specific profiles.

A problem occurs, e.g. if one would migrate a PSM based on EJB to Spring and Hibernate. In this case, the EJB profile first has to be unapplied by removing stereotypes and tagged values, then the model has to be modified structurally to conform the constraint forced by Spring and Hibernate profiles, and finally, stereotypes and tagged values defined by the target profiles have to be attached to the model. We identify that problems of profiles are closely coupled with code generators because of the following reasons:

- Each profile defines many stereotypes and tagged values whose names and possible values are closely related to the platform terminology. For example, ≪`Entity`≫ stereotypes denote EJB entity beans in Fig. 1(a), while they are plain-old Java objects (POJO's) managed by Hibernate in Fig. 1(b). Thus, developers are obliged to learn the platform first, rather than the profile itself.

- Concepts and terms introduced by profiles are technical and separated from the requirements. For example, ≪`Service-Operation`≫ stereotype in Fig. 1(b) means that the *addOrder* operation runs an application logic within a transaction because the result should be transactional. Thus, one can hardly tell, which profile need to be applied and how to elaborate the requirements to models.

- Since profiles often put their own constraint over the UML metamodel, it is not easy to migrate a PSM from one profile to another, even if both of them offer similar functionalities and thus are alternatives for the application. For example, the relationship between *OrderService* class to *BookOrder* class needs to be an association with ≪`Reference`≫ stereotype in Fig. 1(a) and a dependency in Fig. 1(b).

For these reasons, PSMs are unsuitable to build, maintain and reuse for the further software evolution. We find it difficult to deal with a PSM once an application is built, especially when it has to be migrated to another platform. To address these issues, we propose an approach which enables developers to avoid operating with PSMs and code generators directly.

3. ACCURATE APPROACH

In this section we present the ACCURATE approach. The name ACCURATE comes from an acronym for 'A Configurable Code generator Unified with Requirements Analysis TEchniques'. As it implies, requirements play an important role in both PIM modeling and platform decision. The key idea is to capture functional and non-functional requirements into separate artifacts, a PIM and a platform configuration respectively, and join them at the downstream of the development.

Figure 2 illustrates the workflow of the approach. It defines four activities, PIM modeling, platform decision, PIM-to-PSM transformation and code generation. They are carried out by two kind of actors, application designer and requirements engineer, who are responsible for functional and non-functional aspects of the system respectively. During the proposed workflow, models have to run through different stages (e.g. a PIM is transformed into a PSM).

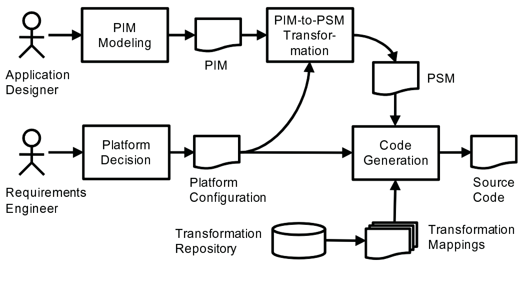


Figure 2: ACCURATE Workflow

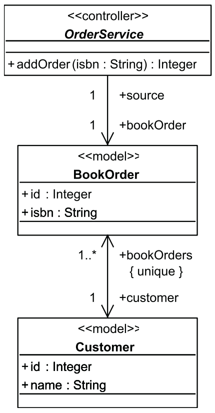


Figure 3: An Example Usage of the ACCURATE Profile

Following subsections explain these activities in terms of their inputs and outputs.

3.1 PIM Modeling

The workflow begins with definition of structure and functionality of the system as a PIM. We defined a platform-independent UML profile, called the ACCURATE profile, to describe PIMs. This profile adopts established concepts defined in the architecture styles as names and semantics of stereotypes, since architecture styles can be considered essentially immutable and independent of any platforms.

Application designers describe PIMs using UML modeling tools (such as MagicDraw [15]), which have support for defining and applying profiles to a UML model. The ACCURATE profile has fewer stereotypes and tagged values so that designers are easily able to learn and use, keeping still expressive enough to specify an application independently of any platform specific details.

Figure 3 illustrates a possible usage of the ACCURATE profile for the well-known *Model-View-Controller* architecture style [6]. According to this style, ≪model≫ classes provide core functionalities of an application domain and propagate changes to ≪controller≫ and ≪view≫ classes, which are responsible for inputs and outputs respectively.

3.2 Platform Decision

In parallel with the PIM modeling activity, requirements engineers communicate with stakeholders around the system to assess quality attributes expected for the system. The output from this activity is a combination of platforms, which usually tends to depend on experience and knowledge of requirements engineers since estimating quality of the system before implementing it is essentially a hard problem.

We assume our approach could be combined with certain requirement analysis and quality estimation techniques, but this topic is out of the scope of this paper due to the limitation of pages.

3.3 PIM Transformation

Once platforms are determined for a system, a PIM can be transformed to a PSM automatically by a model transformation. The output from this activity is a PSM that conforms to the UML profiles for the designated platforms. It can be used directly as an input for the following code generation activity.

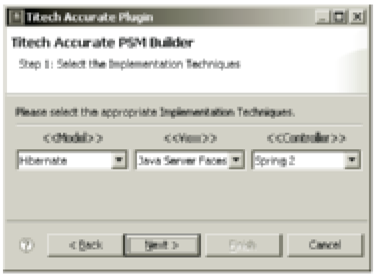
```
/* map a PIM class to a PSM identically */
mapping Class::toPSMClass() : Class {
  name := self.name.firstToUpper();
  isAbstract := self.isAbstract;
  visibility := self.visibility;
  ...
  ownedAttribute := self.ownedAttribute->map
    toProperty()->asOrderedSet();
  ownedOperation := self.ownedOperation->map
    toOperation()->asOrderedSet();
}

/* map a PIM operation to a PSM identically */
mapping Operation::toPSMOperation() : Operation {
  name := self.name;
  type := self.type;
  ...
  ownedParameter := self.ownedParameter->map
    toPSMParameter()->asOrderedSet();
}

/* map a Controller class to a Service class */
mapping Class::toService():Class
inherits Class::toPSMClass
when{
  self.isStereotypeApplied(ACCURATE::controller)
}{
  end {
    result.applyStereotype(Spring2::Service);
  }
}

/* map an operation on a controller class to
   a ServiceOperation */
mapping Operation::toServiceOperation():Operation
inherits Operation::toPSMOperation
when {
  self.class.isStereotypeApplied(ACCURATE::controller)
}{
  end {
    result.applyStereotype(Spring2::ServiceOperation);
  }
}
```

Figure 4: Mappings between the ACCURATE and the Spring2 Profiles

To implement this transformation, we defined mappings between elements of a PIM and a PSM for each supported platform. A transformation can be achieved by a stepwise conversion of all contained elements of the PIM due to the mappings to PSM elements. To define these mappings, we categorized existing stereotypes and tagged values of the profiles for PSMs according to the established concepts used in the architecture styles. Although architecture styles defines typical structure and behavior of the elements, they usually need to be modified due to additional constraints enforced by target platforms.

For example, let's consider a mapping from a PIM element *OrderService* with the stereotype ≪controller≫ (see Fig. 3) to the PSM element *OrderService* with ≪Service≫ and ≪ServiceOperation≫ stereotypes for the Spring Framework (see Fig. 1). Figure 4 shows a part of the mappings specified in the MOF QVT operational language [17]. These mappings create PSM elements from input PIM elements and map ACCURATE stereotypes to Spring ones.

As one might notice, not only the stereotypes and tagged values need to be changed, but it is also required to remove unnecessary elements (such as ≪external≫ stereotyped elements) or modify the structure (e.g. change associations to dependencies) in this transformation.


Titech Accurate Plugin

Titech Accurate PSM Builder

Step 1: Select the Implementation Techniques

Please select the appropriate Implementation Techniques.

<<Model>> <<View>> <<Controller>>

Hibernate Java Server Faces Spring 2

Figure 5: Platform Selection Dialog for the MVC Architecture Style

3.4 Code Generation

Source code for the application based on the platform configuration is generated at the end of the workflow. Here we make use of existing code generator frameworks (such as oAW, AndroMDA) that support various platforms by separating definitions of transformation mappings from their execution engines. Such transformation mappings are often called *cartridges* due to their replaceable character and stored in the transformation repository for reuse (as shown in Fig. 2).

According to the platform configuration determined by the platform decision activity, transformation mappings are chosen from the repository to configure a code generator specific to that platforms. Using a valid PSM from the PIM-to-PSM transformation activity, source code generation can be less error-prone.

4. SUPPORTING TOOLS

We have prototyped a PIM-to-PSM transformation tool as a plug-in for the Eclipse platform. This tool implements transformation mappings using the QVT operational language implemented by the Eclipse M2M [8] project. It offers a user interface to specify a PIM and platform decisions for the system with a wizard-style dialog shown in Fig. 5. Users just have to select appropriate platforms for the ≪`model`≫, ≪`view`≫ and ≪`controller`≫ parts of the target system from drop-down menus.

After the wizard dialog is finished, a PSM and a platform configuration file are generated. This file is used in the following oAW code generator to distinguish, which transformation mapping have to be executed from the transformation repository to generate source code conforming to the designated platforms. As for the PSM-to-PSI transformation, we make use of the oAW code generator framework. One common transformation repository for oAW is the Fornax-Platform, which offers a variety of cartridges to generate application code based on profiled UML models and thus is a possible candidate for the code generator in our tool chain.

5. CASE STUDY

In order to evaluate our approach, we have carried out a case study derived from a possible real-world scenario in which a system is using a specific platform technique. Due to changing requirements of the project, the platform decision was out-dated. As a result, the PSM and PSI have to be regenerated to adopt the new platform decision. The aim of the case study is to show that a platform, developed using the ACCURATE approach, can handle such a situation properly. Furthermore, we are going to argue on the feasibility and benefits of our approach in Sect. 5.2.

5.1 Address Book Example

Consider a company that is implementing an application for managing their customer's addresses using the ACCURATE approach.

At the beginning of the scenario, designers described a PIM and requirements. The ACCURATE profile is applied to the PIM (as shown in Fig. 6). Around the same time, requirements engineers assessed quality attributes of the system and determined to adopt Hibernate for the ≪`model`≫, POJO's for the ≪`controller`≫ and a Swing GUI for the ≪`view`≫. Using the PIM and the platform decision, the ACCURATE plug-in generated an accordant PSM (see Fig. 7(a)). After transforming the PIM into a PSM, the application consists of 28 generated Java classes (six for Hibernate and 22 POJO's). From these 22 POJO's only seven classes have to be implemented manually since the remaining 15 classes are automatically generated interfaces, abstract or implementation classes that don't need to be modified. Besides this, three Hibernate mapping files and a Hibernate property file are generated that also not have to be modified. The PIM-to-PSM transformation took about one minute and the PSM-to-PSI transformation around ten seconds with an average laptop PC (with a Pentium M processor at 1.60 GHz and 1.5 GB of memory) in this scenario.

At some point of the project, the project manager decided to adopt the Spring Framework as a ≪`controller`≫ technology. Since the PIM doesn't not hold any platform specific information by definition, no changes to the PIM have to be made. Using the ACCURATE plug-in again, another PSM conforming to the new platform is generated in about one minute (as shown in Fig. 7(b)). One might notice that Swing is still used as the ≪`view`≫ technology but the PSM elements are mapped to ≪`SpringBean`≫ instead of ≪`JavaObject`≫ this time. Afterwards, the PSM-to-PSI transformation is triggered to regenerate the source code, which took around ten seconds. At this point manual implementation of the missing parts has already been finished. Since oAW doesn't overwrite manual implementation classes during the PSM-to-PSI transformation, the number of newly generated artifact in this second scenario is lower than before. As a result, one interface, abstract and implementation class for each ≪`controller`≫ component was generated. These classes are stored at a different location due to the platform specification. Furthermore, two helper classes for enhanced access to Spring beans and three configuration files are automatically generated. As the final task, the developer has to move the manually implemented code fragments from the outdated ≪`controller`≫ classes to the newly generated ones.

5.2 Discussion

One of the main benefits shown in this case study is that the platform of the application can be switched within a small time period and without modifying the PIM at all. Since the ACCURATE profile is based on architecture styles, which have an essentially platform independent and immutable nature, PIMs using such a profile show improved maintainability and reusability. Thus, they could live on until some functional requirement changes or platform evolutions occur in the future.

Furthermore, in case that new platforms emerge they need to be adopted to our approach, e.g. another implementation technique for ≪`view`≫ classes. In such a case, we only have to define a transformation from our PIM to the PSM of the new platform, as long as this platform conforms to some architecture styles adopted in the ACCURATE approach. Compared to arbitrary PSM-to-PSM transformations like the example shown in Sect. 2, it is rather straightforward to refine PIM concepts to those of PSM and thus most of the transformation can be automated. It has to be mentioned, that our approach expect a code generator together with a PSM defi-

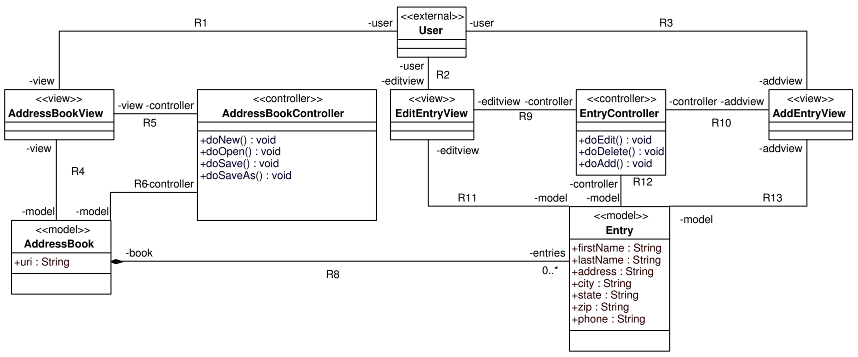


Figure 6: A PIM for the Address Book Example

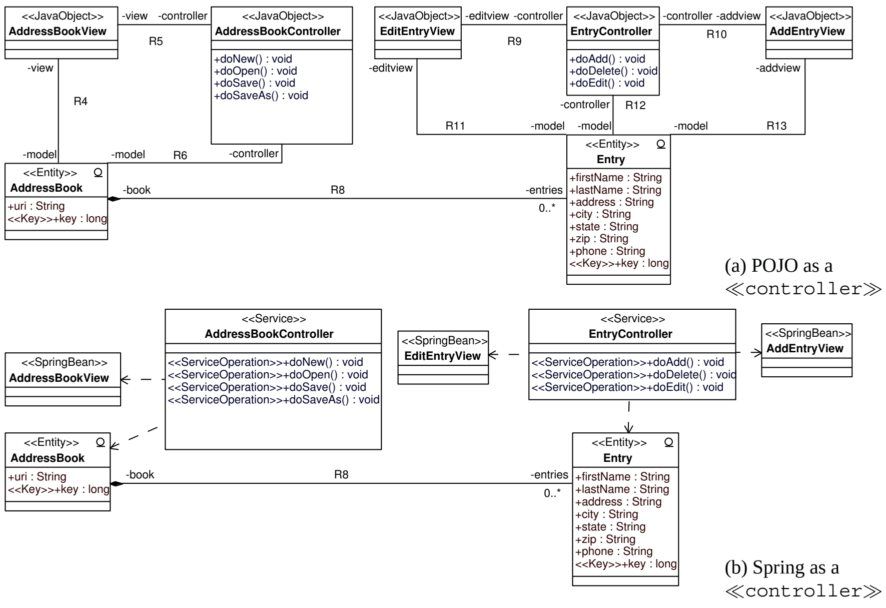


Figure 7: Two PSMs using Hibernate as a ≪`model`≫ and Swing GUI as a ≪`view`≫

nition. Otherwise, we would have to implement the PSM-to-PSI transformation by ourselves.

Another advantage worth mentioning is that handwritten parts of source code are preserved during code regeneration. In the case study, Swing GUI and Hibernate classes are reused, except that their instantiation code (i.e. constructor calls) is replaced by a XML configuration file for Spring. On the other hand, the generated part for ≪`controller`≫ classes are regenerated based on the Spring service components, while handwritten part of the POJO's are left untouched. This means that there are some remaining parts, which have to be migrated manually, even though task can be achieved in a reasonable time due to the size of the handwritten code. We suppose, that generating complete source code from PIM or providing help and guidance for each possible migration are two possible solutions to address these problems.

6. RELATED WORK

There is already some existing work focusing on platform independent modeling and model transformation in a different problem domain. Bezivin et al. [4] propose to use ATL transformation [7] to transform PIMs defined by Enterprise Distributed Object Computing into PSMs for different web service platforms. Billig et al. [5] define PIM-to-PSM transformations in the context of EJB by using QVT [17]. Besides this, some related work define PIMs via UML profiles. Link et al. propose to use GUIProfile to model PIMs and transform them into PSMs [13]. Richly et al. focus on a UML profile to define PIMs for databases [11]. He et al. use template role models together with PIM profiles for templates to design PIMs, which are specific for web applications [12] . Ayed et al. propose a UML profile for modeling platform independent context-aware applications [2]. Lopez-Sanz et al. define a UML profile for service-oriented architectures [14]. Finally Fink et al. combine UML and MOF profiles for access control specifications [9]. As one can notice, there are a lot of approaches, which describe a PIM on a more abstract level than a PSM. Even so, these approaches are still tailored to a specific technology or architecture and thus need some detailed knowledge of the concrete problem domain. Furthermore, the adoption to a different problem domain or architecture such as MVC is hindered due to the specific notations of these PIMs.

7. CONCLUSION AND FUTURE WORK

In this paper, we have stated out some clear problems of MDD approach when it has to change a platform to another. To address this problem, we have introduced an approach called ACCURATE, which consists of a profile for describing PIMs and transformation mappings to bridge a PIM to PSMs of existing code generators. Our approach shows how to specify systems easily without any PSM modeling skills. The approach offers much automation of the development process and thus reducing costs under the pressure of a shorter time-to-market.

Furthermore, a prototype tool is provided, which both assures the integrity during model transformation and offers guidance through the software development process to the user. The current implementation of the tool provides a workable and extendable solution to address the stated problems. However, there are still some enhancements that we would like to adopt to our approach in the near future. These possible extensions can be summarized as follows:

1. **Further evaluation:** This paper focused on applying the ACCURATE approach to the MVC architecture style. Since this is just one possible example for an architecture, we would like to evaluate our approach to a different architecture style (e.g. Pipes and Filters or Blackboard) and on a larger scale to prove the applicability more sustained.

2. **Platform decision models:** As mentioned before, the platform decision can be supported by assessing quality attributes expected for the system. We are going to introduce platform decision models more precisely. The first model we are now focusing on is based on Bayesian networks, which allows to infer platform decisions based on predefined probability distribution metrics.

3. **Interaction with coding:** In theory, complete source code could be generated from a model. But due to unfamiliarity of graphical PIM modeling and immaturity of tool support for MDD at this moment, developers prefer to finish up implementation by complementing or adjusting generated source code in their common programming languages like Java. We would like to adopt oAW recipes to help the developer track the missing parts of the implementation, and hopefully propagate changes in source code (e.g. adding a method) to its originated PIM.

8. REFERENCES

[1] AndroMDA.org. AndroMDA.org - Home. `http://www.andromda.org/`.

[2] D. Ayed and Y. Berbers. UML Profile for the Design of a Platform-Independent Context-Aware Applications. In *MODDM'06: Proceedings of the 1st Workshop on Model Driven Development for Middleware*, pages 1–5, 2006.

[3] J. Bezivin and S. Gerard. A Preliminary Identification of MDA Components. In *GTCMDA'02: Proceedings of the OOPSLA 2002 Workshop in Generative Techniques in the Context of Model Driven Architecture*, 2002.

[4] J. Bezivin, S. Hammoudi, D. Lopes, and F. Jouault. Applying MDA approach for Web service platform. In *EDOC'04: Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference*, pages 58–70, 2004.

[5] A. Billig, S. Busse, A. Leicher, and J. G. Süss. Platform Independent Model Transformation Based on TRIPLE. In *Middleware'04: Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pages 493–511, 2004.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., 1996.

[7] eclipse.org. ATLAS Transformation Language (ATL). `http://www.eclipse.org/m2m/atl/`.

[8] eclipse.org. Model to Model (M2M) Project. `http://www.eclipse.org/m2m/`.

[9] T. Fink, M. Koch, and K. Pauls. An MDA approach to Access Control Specifications Using MOF and UML Profiles. *Electronic Notes in Theoretical Computer Science*, 142:161–179, 2006.

[10] fornax-platform.org. The Fornax-Platform. `http://www.fornax-platform.org/`.

[11] D. Habich, S. Richly, and W. Lehner. GignoMDA: Exploiting Cross-layer Optimization for Complex Database Applications. In *VLDB'06: Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1251–1254, 2006.

[12] C. He, F. He, K. He, and W. Tu. Constructing Platform Independent Models of Web Application. In *SOSE'05: Proceedings of the 2005 IEEE International Workshop on Service-Oriented System Engineering*, pages 85–92, 2005.

[13] S. Link, T. Schuster, P. Hoyer, and S. Abeck. Focusing Graphical User Interfaces in Model-Driven Software Development. In *ACHI'08: Proceedings of the 1st International Conference on Advances in Computer-Human Interaction*, pages 3–8, 2008.

[14] M. López-Sanz, C. Acuña, C. Cuesta, and E. Marcos. UML Profile for the Platform Independent Modelling of Service-Oriented Architectures. *Software Architecture*, pages 304–307, 2007.

[15] No Magic. MagicDraw UML. `http://www.magicdraw.com/`.

[16] OMG. MDA Guide Version 1.0.1. `http://www.omg.org/docs/omg/03-06-01.pdf`, 2003.

[17] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.0. `http://www.omg.org/docs/formal/08-04-03.pdf`, 2008.

[18] openArchitectureWare.org. Official openArchitectureWare Homepage. `http://www.openarchitectureware.org/`.

Evolution of a Domain Specific Language and its engineering environment - Lehman’s laws revisited

Mika Karaila
Metso Automation Inc.
Lentokentänkatu 11
33101 Tampere, FINLAND
+358407612563
mika.karaila@metso.com

ABSTRACT

Automation domain is under continuous change with new requirements. Metso Automation has been one of the first vendors of digital automation systems (1978 Damatic). The last 20 years of development and maintenance of system architecture and a dynamic, flexible engineering environment has enabled us to successfully live with the changes. In this paper, evolution of a domain specific visual system configuration language called Function Block Language (FBL) and a supporting environment is discussed. The evolution is reflected to Lehman's laws. Metso Automation’s solutions for surviving with the implications of the laws are also discussed.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Specialized application languages – *domain specific language, visual language.*

General Terms

Design, Reliability, Experimentation, Languages.

Keywords

Visual Domain Specific Language, Evolution, Software Laws, Software Patterns.

1. INTRODUCTION

A distributed control system (DCS) refers to a control system usually of a manufacturing system, process or any kind of dynamic system, in which the controller elements are not central in location but are distributed throughout the system with each component sub-system controlled by one or more controllers. The entire system of controllers is connected by networks for communication and monitoring.

For building DCSs at Metso Automation, a multi-level architecture is used in MetsoDNA DCS. Controllers use hardware units such as input/output cards (I/O) to connect field devices into a system. Software is located in controllers, I/O cards and field devices. There are different kinds of firmware programs, bus protocol stacks, operating systems and different kinds of tools and databases. An automation system executes programs in real-time in a distributed environment. It can control a small process just some devices or a huge factory with several paper machines, stock preparation and own power plant. One key element is communication between the units. Communication must be deterministic, real-time, robust and scalable.

Function Block Language (FBL), developed at Metso Automation, is a visual programming language for writing real-time control programs for distributed environments. FBL programs are represented as diagrams that will implement application programs. Each diagram typically contains 5-10 smaller application programs, which are loaded into a distributed system. A typical paper manufacturing plant automation is built from 5000 to 10000 FBL programs. They control 15000 input/output connections (I/O). Total amount of small application programs is over 100000.

FBL is part of a bigger product family that has a long life cycle. We will show how in automation domain both FBL and supporting programming environments such as FBL editor and other tools will need effort for their controlled and successful maintenance.

Section 2 introduces FBL and its history in brief. Section 3 briefly introduces Lehman's laws of software evolution [12]. They characterize the ways large software systems tend to evolve. In each subsection, Lehman's laws are discussed with respect to the automation domain, FBL and its programming environment. This section also explains the methods that we use to survive with the evolution. It will explain Metso Automation’s maintenance process and its benefits. The process has been developed to manage the challenges software evolving according to Lehman's laws creates. The key idea is to manage the main principles like working methods and product features and keep the system and domain specific language in balance during evolution. Section 4 will discuss and summarize items introduced

2. Domain Specific Languages

The term domain-specific language (DSL) [2] has become popular in recent years in software development to indicate a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique [11]. The concept is not new. Special-purpose programming languages and all kinds of modeling/specification languages have existed, but the term has become more popular due to the rise of domain-specific modeling (DSM) [10]. Domain-specific languages are 4GL programming languages. Examples include spreadsheet formulas and macros, YACC [5] grammars for creating parsers, regular expressions, Generic Eclipse Modeling System [21] for creating diagramming languages, advanced DSM tool MetaEdit+ [18], Csound [19], a language used to create audio files, and the input language of GraphViz [3], a software package used for graph layout. The opposite of a domain-specific language is a general-purpose programming language, such as C or Java, or a general-purpose modeling language such as UML.

There are advantages in using DSLs. Domain-specific languages allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, programs written using domain experts themselves can understand, validate, modify, and often even develop domain-specific languages. Also, the code is self-documenting in an optimal case. Furthermore, domain-specific languages enhance quality, productivity, reliability, maintainability, portability and reusability [10]. Finally, they allow validation at the domain level.
There are also disadvantages in using domain-specific languages. The cost of designing, implementing, and maintaining a domain-specific language can be very high. Also finding, setting, and maintaining proper scopes can be difficult. There can be a lack of processing capacity compared with hand-coded software. Finally, code can be hard or impossible to debug.

2.1 An overview of FBL

Metso Automation has created its own visual domain-specific language with a supporting FBL framework, i.e. a development environment that supports FBL and its usage.
FBL drawing is a visual program that can be compiled and downloaded into a real-time system. In real-time, the program will, for example, measure a tank's level and control a valve so that tank level will remain at the desired level. An FBL program is a signal flow diagram that contains multiple symbols that are connected by lines. Symbols represent variables or variable references and functions. An FBL diagram is self-documenting, because it is a graphical program that explains the code functionality visually. The generated textual code is multiple pages of text and connections are just text references in multiple places of text. An overview of connections is very hard to understand from the textual format. The visual notation of FBL consists of symbols and lines connecting them. In FBL, symbols represent advanced functions. The core elements of FBL, function blocks, are sub-routines running specific functions to control a process. As an example, measuring the water level in a water tank could be implemented as a function block.
In addition to function blocks, FBL programs may contain port symbols (ports publish access names) for other programs to access function blocks and their values. Function block values are stored in parameters. As an analogy, the role of a function block in FBL is comparable to the role of an object in an object-oriented language. The parameters which can be internal (private) or public, can, in turn, be compared to member variables. An internal parameter has its own local name that cannot be accessed outside the program. A public parameter can be an interface port with a local name or a direct access port with a globally unique name.
FBL programs may also contain external data point symbols for subscribing data published by ports, external module symbols to represent external program modules, and I/O module symbols to represent physical input and output connections. An external data point is a reference to data that is located somewhere else. In distributed control systems, calculations are distributed to multiple calculation units. Therefore, if a parameter value is needed from another module, the engineer has to add an external data point symbol to the program. By using this symbol, data is actually transferred (if needed) from another calculation unit to local memory. An example of an FBL program is depicted in Figure 1. This program is for detecting binary signal change; it will read the state 0 or 1 from the field using the I/O-input symbol BIU8 (A) on the left side. Then the signal is connected to a copy function block (B) and after it to a delay function block (C) that will filter out short time (under 5 second) changes. After the delay filtering, the signal is copied to a port (D) that can be connected to the user interface (E) to show the state in the actual real-time user interface. The state is also stored in the history database (F) that keeps all the state changes for a long time (months or years). The interface port (G) is for the interlocking usage. The signal can be used in other diagrams for interlocking. If the state is for example 1 it can prevent a motor from starting.

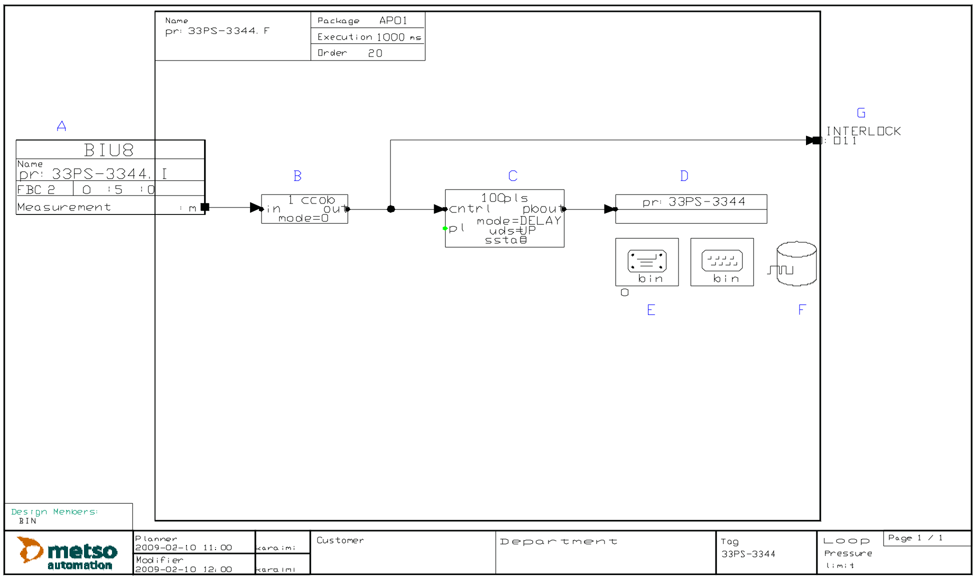


Figure 1 FBL overview.

2.2 Development history of FBL

In 1988, the first release of FBL was introduced to customers. It included all the basic elements: ports, externals, function blocks and I/O modules of the language itself. The code generator was hand written and the implementation contained a pattern based text generator [9] that used output templates. Currently, there are framework tools available for that purpose [17].

The Automation Language is a simple domain specific textual language. The FBL code generator produces Automation Language. FBL has been used from the year 1988. The FBL programming IDE, editor is FbCAD (product name for the Function Block CAD). The Function Test tool was introduced in 1990. It allows the user to debug run-time values in the FbCAD (visual debugging mode). This is not always trivial with DSL because generated code may need its own IDE that can be hard to implement. Function Explorer is a tool for multiple users in a client-server environment. It allows concurrent programming and is an easy way to change parameters for the FBL programs saved in the database. The last major development step was support for "Templates" in the year 2003. Visual templates are domain specific models that can be used to create new FBL program instances efficiently [8]. Another way to support reuse relies on patterns. In the last years, we have detected patterns from visual programs. They are solutions to small problems that can be solved with few symbols.

The development of FBL has had various goals. One of them is to aim at a visual language to make programs understandable and the programming environment easy to use and learn. The user can always add both textual and graphical comments to the program.

The initial setup for developing FBL was the fact that the automation language was too hard to understand and the connection network was impossible to figure from the textual code. The basic architecture separates user interface to its own part and the code generator to its own. In the beginning, the basic user interface was static and very simple. It was then extended, and more dynamics were added, like the visual testing tool. Now the focus of user interface development is on usability issues. Also a lot of new dynamics have been added to fulfill new requirements. Programming effectiveness is one major target. We always try to make programming faster. There are small improvements in the user interface level and in the language level, but the biggest improvement has been the introduction of templates [8]. Templates are reusable domain models that can be instantiated. An instance will need only a set of parameters to work. It has significantly reduced engineering work and made large modifications easy to make. Adding new elements to it have also extended the FBL language itself. Finally, improving the code generation in FBL has raised the quality of the programs. It is more accurate and detects more errors and also gives warnings to users.

If we compare FBL programs created in 1988 and now, the major difference is that they are now bigger and more complex. This is due to the new requirements for higher automation levels.

3. Lehman's laws and evolution of FBL and its supporting engineering environment

In this section, Lehman's laws are revisited in the context of the automation industry domain and in particular the evolution of FBL and its programming environment. Each of eight laws is discussed and the name of the law is the title of the subsection. These results are collected and formed based on 20 years of development history. The connections and network between Metso Automation's own people and customers have given a lot of feedback on FBL and its programming environment. The maintenance process of FBL itself is an iterative process that relies on the feedback system. These processes and methods can be used to gain better results and to manage the evolution FBL.

3.1 Law I: Continuing change

E-type systems must be continually adapted else they become progressively less satisfactory.

The automation domain itself is under change. In addition, also the environment that is used for FBL and its editor and other tools are under change. For instance, the operating system has changed multiple times from UNIX (Xenix, SCO, Ultrix, and HP) to DOS and Windows (NT, XP, Vista) and also the compiler is under change all the time. The CAD platform that is the base of the FBL editor has been changed according to the operating system and needed compiler. The selected CAD platform (AutoCAD) was a market leader. The use of a commercial platform helped a lot, because its development and maintenance was carried out by others. The only major work was to port the FBL user interface (editor) always into the selected release. The selection process was guided by a technology roadmap. This kind of preplanning gave development teams time to prepare for new things in advance.

This does not directly affect FBL itself, but the editor and the code generator both require major maintenance work. There is a compatibility requirement; life-time cycles are demanding in automation domain. FBL editor changes according to the style of the CAD platform and operating system. The change in the visual appearance is significant if we compare the very first 640 x 480 resolution to the current 1024 x 768 one. Outlook is also improved by new better fonts and more colors that are used today. Actual FBL improvements have been mostly visual.

As Lehman's first law indicates, resisting changes is not a fruitful solution in the long run. Instead, we have chosen to live with the changes and upgrade environment. The software environment changes in the domain create needs for changes in FBL and its tools: the domain specific language must evolve with the environment.

3.2 Law II: Increased complexity

As an E-type system evolves, its complexity increases unless work is done to maintain or reduce it.

Metso Automation DCS size grows both hardware & software. Also other additional functions make the system more complex.

As an example, new I/O-cards are needed and they include new features and more channels. FBL language supports changes, e.g. new symbols can be created into FBL. Some of these are typically similar to existing ones, like new function blocks. New function blocks do not extend FBL, but give new features for programming. This was seen already 20 years ago. The internal architecture of the code generator was built to be generic and the variation point was built into symbol level.

There are intelligent devices with new communication protocols evolving which will require support. The Foundation Fieldbus (FF) [4] integration, for instance, needed its own FBL symbols. The code generation was extended to support FF configuration. This required new semantics. This was mainly solved by the

generic part, only the connection solver needed a special algorithm and the 'output-printer' was extended for FF domain with new C++ classes. Other integrated protocols are Profibus DP [16] and OLE [13] for Process Control (OPC) [15]. They, however, did not need such big integration work for FBL.
In the FBL language level complexity is isolated to its own symbols. The code generator architecture does not need in a typical case any changes. The increased complexity is isolated into FBL editor and code generation as configurable extensions. In this way new protocol specific variations can be integrated by settings and they will not need code changes into the FBL tools every time. A typical way to reduce complexity is abstraction and capsulation, but in cases where this is not possible it is good to first identify variation points and then locate them to selected places in architecture.

3.3 Law III: Self regulation

E-type system evolution process is self regulating with distribution of product and process measures close to normal.

In the automation domain we cannot release a new build each week or month. Customers cannot shut down factories so often. A normal case is to have one planned shutdown each year, sometimes perhaps only twice a year.
In distributed automation system architecture allows that parts can be turned off and on. In this way non-critical parts can be updated/upgraded or even replaced while the process is running. Usually it is a broken device or I/O-card that must be replaced.
Same modularity can be seen in FBL language level. An FBL program can be downloaded into the system in runtime without any disturbance. The modularity makes it possible to download small application programs into the running factory without interruptions.
The evolution process that requires new technologies works according to Moore's law [14]. But in automation domain a new technology, for example a new operating system, is taken into use after careful consideration and after other industry experiences. Technological steps are taken in 2-3 year intervals. As an example, FBL editor and operating system are upgraded with that interval. A conservative attitude and caution normalize technological evolution.

3.4 Law IV: Conservation of organizational stability

The average effective global activity rate in an evolving E-type system is invariant over product lifetime.

In the automation domain you have to know something about field devices, process, and electronics. The automation system architecture and teams are formed in the same way (logical structures are similar, c.f. Conway's law [1]). Development and project organizations have been structured in the same way for about the last 20 years. The team cannot change many persons at the same time because the learning process takes time. Nobody can have all the knowledge but a good programmer must understand the domain. It usually takes months to start to understand the whole automation domain from the controller level to the device level.
A small core team is an example of good practice that allows smooth FBL development and maintenance. A challenging environment and continuous learning keep these people pleased. The needed domain knowledge that requires multi-talented people will help in keeping organizational stability.

3.5 Law V: Conservation of familiarity

As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.

The principles, namely business rules in the domain, are very old in distributed systems. As new communication protocols are integrated into the system, they will have the following typical characteristics like determinism and robustness. These kinds of facts always keep architectural solutions very stable. There are architectural level design patterns that give good solutions to these problems.
The abstraction level in the FBL language is selected to hide unnecessary parts from the end user. The needed parameters are asked from the user and the semantics and the basic layout of communication symbols have remained the same since 1989. In the FBL language the basic symbols are still almost identical.
We have to keep all new extensions somehow similar to existing ones. FBL symbol editor and FBL editor are integrated and mainly old symbols are handmade. The FBL editor has same logical operations for all similar symbols. In this way, the learning process is easier and more logical for an engineer who will use FBL. As people are conservative and do not like very big changes it helps to keep things familiar.

3.6 Law VI: Continuing growth

The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.

Automation systems are typically growing. The application programs that we have delivered to factories are growing. In the hardware architecture the old hardware was VME based Motorola 68030 processors with 2 Mb Memory. Now we use Intel based Pentium with 256 Mb Memory. Also the communication bus speeds have improved from 2 Mb/s to 100 Mb/s. In the hardware level the growth is seen clearly.
We have a reuse library that contains most of the delivered projects. We have measured from the library statistics that the average amount of function blocks in the FBL program has increased from 20 to 30 in the last 10 years (cf. Table 1).

Table 1 Project Function Blocks and Complexity growth.

Project	Average number of Function Blocks	Complexity
A 1999	15	8
B 1999	27	3
C 1999	15	5
D 1999	7	2
E 1999	28	9
F 2008	34	19
G 2008	25	10
H 2008	30	12
I 2008	28	15

The project size has grown from 1000 to 5000 application programs. This is partly due to the technology change. Field devices are more intelligent and they are connected by bus into the system. Instead of having signals connected by traditional wires there are multiple software signals coming from one physical connection. But each software signal still needs its own handling, which causes growth. This all means that the total amount of program code is five times more than 10 years earlier (1999 --> 2008).
We can get these statistics of FBL usage easily because the same working methods are used in each customer project. One part of the customer project process is to archive it.
The engineering tools are also growing. We can measure from the version control system statistics that will show the FBL code generator & DB-adapter growth from 2000 to the current year. 2008 has been about 10 kLOC, which is in average 1 kLOC/year (numbers shown in Table 2). The statistics show that FBL itself has grown during 2000-2008 with about 600 new function blocks and other symbols, the average being 75 new symbols per year.
We have thus observed also growth in the FBL language itself, not only in its programming environment. The author of this paper is not aware that Lehman's laws have been earlier discussed in the context of programming languages. They are, however, widely discussed in the context of large software systems. Continuous growth of FBL itself is interesting and due to its increasingly broad use in different contexts and by different customers. Moreover, we assume that such growth is not typical for general purpose languages, but can be more natural for domain-specific ones.

Table 2 Code generator and DB adapter size and growth.

Program	Code and Lines in Year 2000	Codes and Lines in Year 2008
Code Generator	35999	44304
DB adapter	20642	31986

These numbers show that increased functionality increases the code in the application layer (not in the system core). In the system core, the increase comes from the supported hardware, operating systems and new communication protocols. We can manage the growth because it is isolated into selected places. The variation points are designed and the solution is to use data centric generation (usually more symbols needed). The code generator core part is very stable. The initial number of symbols was approx. 500 and now it is over 1600 symbols. A good architecture helps in managing the growth. The amount of code is not growing, instead, the growth is at data level.
A very long life-cycle and evolution has not yet affected to the meta-model. The original meta-model is still used. The architecture separates extensions into symbols, and the code generator is still quite compact. The language rules and semantics are fine-tuned by the code generator.

3.7 Law VII: Declining quality

The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

The previous laws, like continuous change, increased complexity and continuing growth, are easily causing problems in quality control. Also all new features, operating system/hardware changes and new protocols can cause new bugs.
The basic architecture isolates modifications. It will also keep the system robust because a bug in one part will not crash the whole system. In the language level, FBL helps in regression testing because it can be used in different environments and different versions. All old FBL programs should be compatible upwards. This kind of FBL interoperability helps in testing. The same FBL program can be used and code generator results can be used for comparing and validating that the system is still working in the same way. Interoperability and compatibility can be used in regression testing to help in quality assurance.

3.8 Law VIII: Feedback system

E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

New communication methods like the internet and email allow customers to contact vendors much easier. The information collecting process uses data from multiple sources. The feedback process is shown in Figure 2. Wishes for new features and minor changes come from the testing and support contacts. All the testing defects are reported and stored in a database. This database is actually a huge diary that contains events that are caused by programmers or designers. The support contacts from the customers or own personnel are also stored in the database. These tools are now integrated so that the user can link and create cards just by one click. So for the development and maintenance all the defects are collected into one database. In this way, it is much easier to prioritize errors and decide who should fix and test the error and when. Information processing is now easier and project managers can focus on those errors in priority order. This makes the focus setting easier and the error handling is up to date all the time.
The process is now organized and made formal. It allows us to have the feedback system running 24/7 and also check that each case is handled. We archive and analyze all the feedback so that it helps us to improve the quality of the products. The amount of feedback issues have grown from some hundreds to over three thousand during the last 20 years. The actual reason comes from the fact that earlier issues were handled more freely. Formalized feedback entering was started at the end of the year 2004. This made it more visible and easier to statistically handle all issues. In numbers this means: bug reports over 1300/year, support cases 1800/year, dissatisfactions over 100/year and ideas 200/year. These issues concern engineering, user interface, controls and hardware parts. Most of the issues are not critical, but they are focusing mostly on user interface and usability issues today. We formalize, control and analyze all collected feedback to really improve both product quality and product features.
The whole feedback framework is made to help, link and reuse information more easily. Also tracking and testing is managed through the process. The process is more transparent and tracking from initiator, coder, and tester to final version report is possible. Each bug report or feature request has its own number and those can be selected to a version report. This makes the quality of the process better. Different views into bug records make it possible to filter and find not handled records. One way to first categorize a bug is to use architecture. The component level can be used to

assign a bug for fixing. In the same way, the project manager and project number can be found and used in the process.

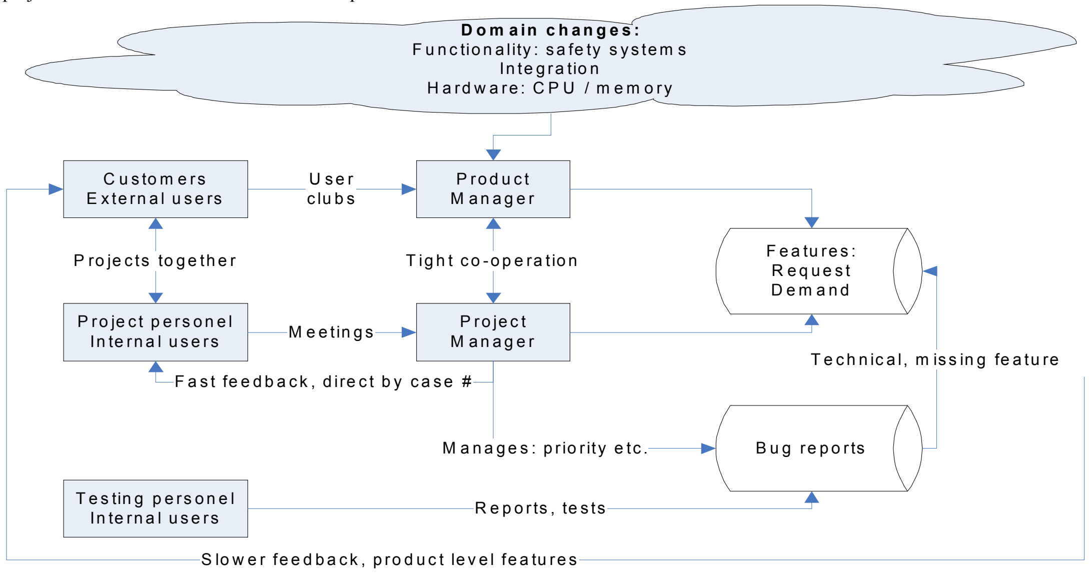


Figure 2 FBL development processes and feedback channels.

The feedback process is a multi-level, multi-feedback system and making it formal will help in tuning it.
FBL improvements that are collected are currently focused on symbols. The symbols´ size is too big or some feature is missing from the function block. Another found improvement comes from the size of the FBL programs. New navigation and intelligent find actions are needed in the FBL editor. Also a more context sensitive user interface is required. These kinds of features exist for example in Microsoft Visual Studio.

3.9 Patterns & idioms in Domain Specific Language

As discussed above, Lehman’s laws for software evolution apply for the FBL environment, and partly to the FBL language itself. In addition, certain patterns / idioms seem to occur in FBL programs and eventually turn into common practices. FBL template models form patterns that are heavily reused. In the same way smaller coding patterns (idioms) exist in FBL. For instance, idioms in FBL can solve a problem that can be seen in runtime behavior. Next two small function block level examples are presented, identified from the FBL programs.

The first idiom is negation. A binary signal with a value 1 or 0 can be converted with one function block “NOT”, but more common is to use “XOR” for that because if negation is not needed, XOR can be turned off by setting the input to 0. This can be done at run-time which is a very good feature in a real-time environment. The original use of NOT function block is thus not any longer so common because if the logic is designed first in the wrong way, the designer must remove the symbol and connect the signals again.

The second pattern is alarm masking. In many cases the FBL program contains function blocks that will generate an alarm. In process control there are abnormal situations like starting the process or shutting it down. In these cases there can be off limit values in the measurements. It is typical to suppress these by masking the alarm signals for a certain time like 10 to 30 seconds. A time alarm is needed because there is most probably some real problems in the process.

As design patterns are identified in traditional programming languages and there are architectural level patterns, it is natural to also find them in DSL. Besides supporting FBL programmers, they can partly support the maintenance and evolution process of the language.

4. SUMMARY

In this paper, the evolutionary history of FBL, a language used for implementing automation control programs, and its programming environment has been discussed.
The use of FBL is growing. Interestingly, we have observed that the projects that are using visual programming are very well on schedule. Component reuse is the first step in efficient programming [6, 7, 8 and 20]. Metso Automation’s future work will concentrate on patterns and template maintenance and we will look for extending FBL to integrate more advanced functions. Development agility is the part of the process that has modified FBL and the tools to be flexible. According to our experiences at Metso Automation, language development is a fascinating and dynamic challenge but it requires a well-managed maintenance and evolution process. We have also noticed that the key to such a successful and controlled evolution process is in collecting feedback from different stakeholders and in storing, managing, and using it to further enhance the language. The management

must have a very large product view and good knowledge about used techniques.
These experiences are limited to the automation industry and in particular to FBL and our experiences at Metso Automation. Therefore, we do not claim all the results can be directly generalize to e.g. general purpose programming languages. However, we feel that the process improvement and methods to live with Lehman's laws can be adapted to other cases and software maintenance. In a dynamic environment, it is very important to manage the maintenance and evolution processes.
One success factor has been that we control the maintenance and evolution process with iteration. An essential factor for the success has been a feedback handling mechanism that gives us priorities and new ideas for further development. Another success factor is architecture that is still dynamic and flexible.
The management of development and maintenance processes help in evolution. Both processes have gone through improvements and generations.

ACKNOWLEDGMENTS

The author would like to thank Tarja Systä for her comments and support in writing. This article was written at Tampere University of Technology.

5. REFERENCES

[1] Conway, M.E. 1968 How do Committee's Invent, Datamation, 14 (5): 28-31.

[2] Deursen, A. , Klint, P. and Visser, J. 2000 *Domain-Specific Languages: An Annotated Bibliography*, ACM SIGPLAN.

[3] Ellson, J. and Gansner, E. and Koutsofios, E. and North, S.C. and Woodhull, G. 2002, Graphviz— Open Source Graph *Drawing Tools Springer Berlin / Heidelberg, Volume 2265/2002, 594-597.*

[4] Foundation Fieldbus http://www.fieldbus.org/

[5] Johnson, S. C. 1975 Yacc: Yet Another Compiler-Compiler. Compiler, *Computing Science Technical Report No. 32, , Bell Laboratories, Murray Hill, NJ 07974*

[6] Karaila, M. and Leppäniemi, A.2004 Multi-Agent Based Framework for Large Scale Visual Program Reuse, *IFIP, Volume 159/2005, 91-98.*

[7] Karaila M., Systä T. 2005 On the Role of Metadata in Visual Language Reuse and Reverse Engineering – An Industrial Case *Electronic Notes in Theoretical Computer Science, 2005, Volume 137, Issue 3, 29-41.*

[8] Karaila, M. and Systä, T. 2007 Applying Template Meta-Programming Techniques for a Domain-Specific Visual Language--An Industrial Experience Report, ICSE 2007.

[9] Kastens, U. PTG: Pattern-based Text Generator. v1.1

[10] Kelly, S. and Tolvanen, J-P.2008 *Domain-Specific Modeling Wiley-IEEE Computer Society Press, 448.*

[11] Korhonen, K. 2002 A case study on reusability of a DSL in a dynamic domain, *2nd OOPSLA Workshop on Domain Specific Visual Languages.*

[12] Lehman, M.M. ,Ramil, J F. ,Wernick, P D. ,Perry, D E. and Turski, W M. 1997 Metrics and laws of software evolution -The Nineties View Software, *Proc. of the 4th International Symposium on Software Metrics.*

[13] Microsoft OLE. http://support.microsoft.com/kb/86008

[14] Moore, G. 1965 Moore's law.

[15] OPC communication http://www.opcfoundation.org/

[16] Profibus http://www.profibus.com/

[17] Schmidt, C. and Kastens, U. and Cramer, B. Using DEViL for Implementation of Domain-Specific Visual Languages. University of Paderborn.

[18] Tolvanen, J-P and Pohjonen, R. and Kelly, S. 2007 Advanced Tooling for Domain-Specific Modeling: MetaEdit+., Computer Science and Information System Reports, Technical Reports, TR-38, University of Jyväskylä, Finland 2007, ISBN 978-951-39-2915-2.

[19] Vercoe, B. 1992 A Manual for the Audio Processing System and Supporting Programs with Tutorials.

[20] Vyatkin, V. and Hanish, H-M. 2005 Reuse of Components in Formal Modeling and Verification of Distributed Control Systems *ETFA 2005. 10th IEEE Conference on Publication Date: 19-22 Sept. 2005 Volume: 1 On page(s): 129 - 134, 2005, Volume 1, 129-13.*

[21] White, J. and Douglas C. Schmidt, 2007 A. N. E. W. Introduction to the Generic Eclipse Modeling System, *Eclipse Magazine, Vol. 6, 11-19.*

Automatic Domain Model Migration to Manage Metamodel Evolution

Daniel Balasubramanian, Tihamer Levendovszky,
Anantha Narayanan and Gabor Karsai
Institute for Software Integrated Systems
2015 Terrace Place
Nashville, TN 37203
{daniel,tihamer,ananth,gabor}@isis.vanderbilt.edu

ABSTRACT

Metamodel evolution is becoming an inevitable part of software projects that use domain-specific modeling. Domain-specific modeling languages (DSMLs) evolve more frequently than traditional programming languages, resulting in a large number of invalid instance models that are no longer compliant with the metamodel. The key to addressing this problem is to provide a solution that focuses on the specification of typical metamodel changes and automatically deduces the corresponding instance model migration. Additionally, a solution must be usable by domain experts not familiar with low level programming issues. This paper presents the Model Change Language (MCL), a language and supporting framework aimed at fulfilling these requirements.

1. INTRODUCTION

Model based software engineering has been especially successful in specific application domains, such as automotive software and mobile phones, where software could be constructed, possibly generated from models. A crucial reason for this has been the tool support available for easily defining and using domain specific modeling languages (DSMLs). However, the quick turnover times required by such applications can force development to begin before the metamodel is complete. Additionally, the metamodel often undergoes changes when development is well underway and several instance models have already been created. When a metamodel changes in this way, it is said to have *evolved*. Without supporting tools to handle metamodel evolution, existing instance models are either lost or must be manually *migrated* to conform to the new metamodel.

The problem of evolution is not new to software engineering. In particular, databases have been dealing with schema evolution for several years. While there have been attempts to extend these techniques to model-based software [4], two characteristics of DSMLs suggest that a dedicated solution is more appropriate. First, metamodels tend to evolve in small, incremental steps, implying that a model evolution tool should focus on making these simple changes easy to specify. This also means that a large portion of the language is unaffected between versions: an ideal solution should leverage this knowledge and require only a specification for the portion that changes and automatically handle the remaining elements. On the other hand, complex changes do sometimes occur, so a mechanism for these migrations must also be available. The second point in favor of a dedicated model migration tool is that domain designers and modelers are often not software experts, which means a solution should use abstractions that are familiar to these users and avoid low level issues, such as persistence formats. Ideally, the modeler should be able to use the same abstractions to build models, metamodels and evolution specifications.

Our previous work with sequenced graph rewriting [1] provided some insight into the balance between expressiveness and ease of use. We have found that a general purpose transformation language tends to be cumbersome for the mostly minor changes present during metamodel evolution. Thus, we have designed a dedicated language called the Model Change Language (MCL) used to specify metamodel evolution in DSLs and migrate domain models. The rest of this paper describes MCL and is structured as follows. Section 2 presents further motivation and background terminology. Section 3 describes the overall design of MCL, while the implementation is presented in Section 4. Related work is found in Section 5, and we conclude in Section 6.

2. MOTIVATION AND BACKGROUND

Our primary motivation was drawn from experience with several medium and large DSMLs that continually evolved. The large number of existing instance models made manual migration impractical. For very simple language changes, such as element renamings, we found that XSLT was an acceptable solution. [11] describes a language capable of generating XSL transforms that are applied sequentially, which increases the expressiveness of the evolution, but requires the user to define the control structure and order of evaluation explicitly. Additionally, we occasionally faced more complex changes, for which XSLT was not sufficient. For these changes, our graph transformation language, GReAT [1], provided a powerful alternative, but its model migration specifications were too verbose for two primary reasons. First, when a metamodel element changes, the migration rule should be applied to all instance model elements of that type, regardless of where they are located in the model hierarchy. Second, metamodels tend to evolve in small, incremental steps, in which the majority of the elements stay the same. Together, these two points imply that a model migration tool should:

1. Contain a default traversal algorithm.

2. Automatically handle non-changed elements.

We incorporated both of these ideas into our design. Our essential hypothesis is that evolutionary changes on the modeling language will be reflected as changes on the metamodel. When the modeling language is evolved, the language designer has to modify the metamodel that will now define the new version of the language. The key observation here is that metamodel changes are explicit, and these changes are used to automatically derive the algorithm to migrate the models in the old modeling paradigm to the models compatible with the new version of the paradigm. We make an essential assumption: changes performed on the metamodel are known and well-defined, and all these changes are expressed in an appropriate language. We designed such a language, which we call the Model Change Language (MCL). We now briefly describe relevant background terminology.

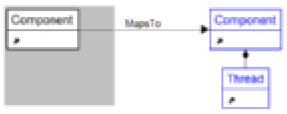
A metamodel $\mathbf{M}_L$ defines a modeling language $\boldsymbol{L}$ by defining its abstract syntax, concrete syntax, well-formedness rules, and dynamic semantics [1]. Here, we are focusing on the abstract syntax of the modeling paradigm. There are various techniques for specifying the abstract syntax for modeling languages, and the most widely used is the Meta-object Facility (MOF) [10], but for clarity here we will use UML class diagrams. The examples in this paper use UML class diagrams with stereotypes indicative of the role of the element, such as *Model* (a container), *Atom* (an atomic model element) or *Connection* (an association class) - but they may be understood as simple UML classes. Note that the actual models can be viewed as object diagrams that are compliant with the UML class diagram of the metamodel.

3. THE MODEL CHANGE LANGUAGE

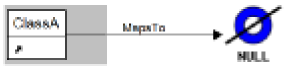
The Model Change Language (MCL) defines a set of idioms and a composition approach for the specification of the migration rules. The MCL also includes the UML class diagrams describing both the versions of the metamodel being evolved, and the migration rules may directly include classes and relations in these metamodels. MCL was defined using a MOF-compliant metamodel. For space reasons we cannot show the entire metamodel, rather we introduce the language through examples. Note that MCL uses the metamodel of the base metamodeling language, and MCL diagrams model relationships between metamodel elements. For a more in-depth look at MCL, please see [2].

The basic pattern that describes a metamodel change, and the required model migration, consists of an LHS element from the old metamodel, an RHS element from the new metamodel, and a *MapsTo* relation between them (stating that the LHS type has "evolved" into the RHS type). The pattern may be extended by including other node types and edges into the migration rule. The node at the left of the *MapsTo* forms the context, which is fixed by a depth first traversal explained in Section 4. The rest of the pattern is matched based on this context. The *WasMappedTo* link in the pattern is used to match a node that was previously migrated by an earlier migration rule. For the sake of flexibility, it is possible to specify additional mapping conditions or imperative commands along with the mapping. This basic pattern is extended based on various evolution criteria, as explained below.

The MCL rules can be used to specify most of the common metamodel evolution cases, and automate the migration of instance models necessitated by the evolution of the metamodel. The core syntax and semantics is rather simple, but for pragmatic purposes higher-level constructs were needed to describe the migration. We have identified a number of categories for metamodel changes based on how metamodels are likely to evolve and created a set of MCL *idioms* to address these cases. These idioms may also be composed together to address more complex migration cases. We will describe a number of these idioms next. We first introduce the representative patterns.



(a) Adding a new element



(b) Deleting an element

Figure 1: MCL rules for adding and deleting elements

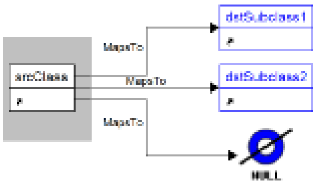
3.1 Adding Elements

A metamodel may be extended by adding a new concept into the language, such as a new class, a new association, or a new attribute. In most cases, old models are not affected by the new addition, and will continue to be conformal to the new language, except in certain cases. If the newly added element holds some model information within a different element in the old version of the metamodel, the information must be appropriately preserved in the migrated models. In fact, this falls under the category of "modification" of representation, and is described further below.

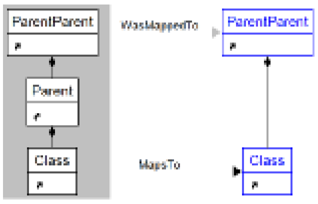
If the newly added element plays a role in the well-formedness requirements, then the old models will no longer be well formed. The migration language must allow the migration of such models to make them well formed in the new metamodel. For instance, suppose that the domain designer adds a new model element called *Thread* within a *Component* - and adds a constraint that every *Component* must contain at least one *Thread*. The old models can then be migrated by creating a new *Thread* within each *Component*, as shown in Fig. 1(a). The LHS or 'old' portion of the MCL rule is shown in a greyed rectangle for clarity in this and all subsequent figures.

3.2 Deleting Elements

Another typical metamodel change is the removal of an element. If a type is removed and replaced by a different type, it implies a modification in the representation of existing information and is handled further below. On certain occasions, elements may be removed completely, if that information is no longer relevant in the domain. In this case, their representations in the instance models must be removed. The removal of an element is specified by using a "NULL-Class" primitive in MCL, as shown in Fig. 1(b). This rule states that all instances of *ClassA* in the model are to be



(a) MCL rule for subclasses



(b) Changing containment hierarchy

Figure 2: MCL Rules for Subclasses and Hierarchy

removed. Removal of an object may also result in the loss of some other associations or contained objects.

3.3 Modifying Elements

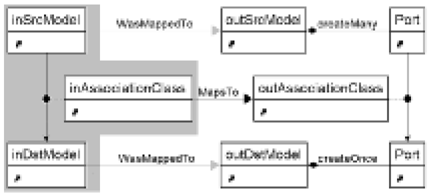
The most common change to a metamodel is the modification of certain entities, such as the names of classes or their attributes. The basic MapsTo relation suffices to specify this change. The mapping of related objects is not affected by this rule. If other related items have also changed in the metamodel, their migration must be specified using additional rules.

Another type of modification in the metamodel is adding new sub-types to a class. In this case, we may want to migrate the class's instances to an instance of one of its sub-types. Fig. 2(a) shows an MCL rule that specifies this migration. The subtype to be instantiated may depend on certain conditions, such as the value of certain attributes in the instance (this is encoded within the migration rule using a Boolean condition for each possible mapping). The rule in Fig. 2(a) states that an instance of srcClass in the original model is replaced by an instance of dstSubclass1 or dstSubclass2 in the migrated model, or deleted altogether.

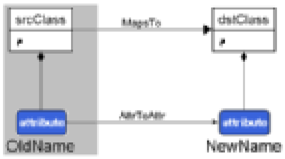
3.4 Local Structural Modifications

Some more complex evolution cases occur when changes in the metamodel require a change in the structure of the old models to make them conformant to the new metamodel. Consider a metamodel with a three level containment hierarchy, with a type *Class* contained in *Parent*, and *Parent* contained in *ParentParent*. Suppose that this metamodel is changed by moving *Class* to be directly contained under *ParentParent*. The intent of the migration may be to move all instances of *Class* up the hierarchy. The MCL rule to accomplish this is shown in Fig. 2(b) (the *WasMappedTo* link is used to identify a previously mapped parent instance).

Note that this rule only affects Class instances. The other entities remain as they are in the model. Any *Parent* instances within *ParentParent* remain unaffected. If *Class* contained other entities, they continue to remain within *Class*, unless modified by other MCL rules.



(a) Rerouting associations



(b) Migrating attributes

Figure 3: MCL Rules for Associations and Attributes

3.5 Idioms and Complex Rules

Based on the descriptions given above, we created a set of idioms that capture the most commonly encountered migration cases. Fig. 3(a) shows a more complex idiom for rerouting associations. The specific case shown here is rerouting associations through ports that are contained model elements under some container. In the old language we had *inAssociationClass*-es between *inSrcModel*-s and *inDstModel*-s, and the new language the same association is present between the *Port*-s of the *outSrcModel* and *outDstModel* classes that were derived from the corresponding classes in the old model. The WasMappedTo link is used to find the node corresponding to the old association end. For the correct results, the new association ends must be created before the MapsTo can be processed for the association, and this is enforced by the use of the WasMappedTo link.

The MCL also provides primitives to specify the migration of attributes of classes in the metamodel. Attributes may be mapped just like classes, and the mapping can perform type conversions or other operations to obtain the new value of the attribute in the migrated model. Fig. 3(b) shows an MCL rule for migrating attributes.

In addition to the idioms listed so far, the tool suite for model migration supports additional idioms to handle other common migration cases. Fig. 4 shows the idiom for adding a new attribute to some class in the metamodel. If the newly added attribute is mandatory, then it must be set in old models that did not have the attribute. A default value can be added for the attribute in the idiom, or a function may be added to calculate a value for the new attribute based on the values of other attributes in the instances. The idiom for deleting an attribute is similar to the case of deleting classes and is not shown due to space constraints. Fig. 5(a) shows an idiom for the case when an inheritance relationship has been removed from the metamodel (the portion above the dashed line is not part of the rule, but shown for clarity). If the derived class had an inherited attribute, this will no longer be present in the migrated model, and must therefore be deleted.

Fig. 5(b) shows an idiom for changing a containment relationship in the metamodel. This is a variation of the idiom shown earlier in Fig. 2(b), for a more generic case. This idiom also introduces a generic primitive called "Navigate". It can be used to locate objects in the instance model by following a navigation condition, which is an iterator over the graph. Starting from the object on the left end of the Navigate link, this object is used to determine the new parent in the migrated model. Fig. 6(a) shows an idiom for merging two classes in the metamodel into a single class, possibly adding an attribute to record its old type. This is effected using two migration rules (shown separated by a dashed line). The migration rule can encode a command that will set the value of the attribute based on its original type. Fig. 6(b) shows an idiom for the case where an association in the metamodel is replaced by an attribute on the source side of the metamodel. This is accomplished by mapping the association to a "null" class (similar to the 'delete class' case) and adding a new attribute on the destination side.

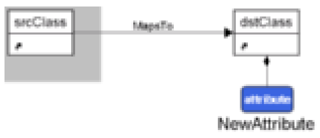
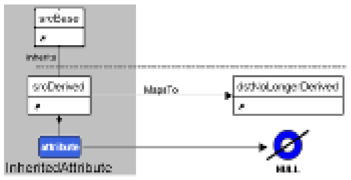
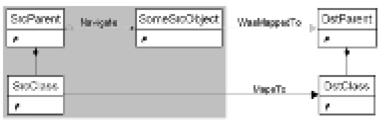


Figure 4: MCL Rule for Attribute Addition



(a) Delete inheritance relationship



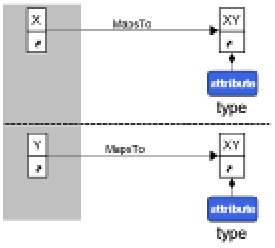
(b) Change containment relationship

Figure 5: MCL Rules for Inheritance and Containment

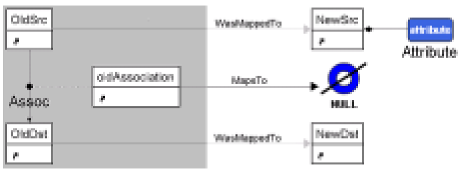
These idioms may also be composed together to accomplish more complex evolutions. The following section presents the details of the MCL implementation.

4. IMPLEMENTATION OF MCL

Our model migration approach consists of three aspects. The first is a complete tree rewrite based on the depth-first traversal of the input model. The second aspect consists of a set of migration rules that specify the rewriting of the model elements (nodes) whose type has changed in the metamodel. The third aspect is a delayed rewrite approach that uses lazy evaluation for the rewriting of nodes that cannot be immediately processed. These are explained in detail below. The migration algorithm maintains a map of the node instances migrated so far (mapping a node in the old version model to its corresponding node in the migrated model), which we call the *ImageTable*, allowing the use of previously mapped nodes in other migration rules. We found that this approach best suited our needs for model evolution, as it simplified the specification and execution of the migration rules. The pattern matching effort required by these rules is limited, while allowing the co-existence of different versions of the model.



(a) Merge classes



(b) Replace association with attribute

Figure 6: MCL Rules for Merging Classes and Replacing Associations

Depth-first traversal algorithm

```
ModelMigrate( oldModel )
  call TraverseTree( oldModel.RootFolder )
  if delayQueue.length != 0 then call ProcessQueue

TraverseTree( node )
  call MigrateNode( node )
  foreach childnode in node.children do
    call TraverseTree( childnode )
```

Depth-first traversal and rewrite. The tree rewrite starts at the root node (*RootFolder*) of the input model, creating a corresponding root node for the migrated (output) model. It follows a depth-first traversal of the input model based on its containment relationships, while creating the output model in the same order. Each node is migrated either by (1) a default migration which creates a 'copy' in the output model, or by (2) executing the migration rule specified for its type. Some migration rules may not be executed immediately and are queued and handled later.

Migration Rules. Typically, when a metamodel evolves, only a small number of the types and relations defined in the metamodel are changed. For the unchanged types, the default ‘copy’ operation suffices in the tree rewrite described above. For the cases where the type has changed, a migration rule is used to specify the actions necessary to migrate an instance of that type into the output model.

Migration rules are specified using MCL as described above in section 3. An MCL rule is specified for a particular (node) type in the metamodel, and consists of a pattern which may involve other node types, a *MapsTo* relation that specifies how the node type is migrated, optional *WasMappedTo* relation(s), and additional imperative *commands* and *conditions* to control node creation. The *commands* are imperative actions executed during node creation, and *conditions* are Boolean expressions that control whether the migration is allowed to happen. The *WasMappedTo* relation specifies a node instance in the output model that was previously migrated corresponding to a certain node instance in the input model (maintained in the *ImageTable*). The migration of a node begins by finding a migration rule for that node type. With the node instance as context, the rest of the rule elements are matched by matching the appropriate nodes in the input model. If the match is not successful because the *WasMappedTo* relation is not satisfied (yet), the node is added to a queue to be processed later. Otherwise, the specified node is created in the migrated (output) model, and the depth-first traversal continues.

Migration algorithm for a Node

```
MigrateNode( node )
  let rule = FindMigrationRule( node.type )
  if rule == null then call DefaultMigrate( node )
  else call ExecuteRule( rule, node )

FindMigrationRule( nodetype )
  find rule in ruleSet where mapsTo.LHS.type = nodetype
  if found return rule else return null

DefaultMigrate( node )
  let newtype = node.type
  if newtype not in newMeta.types
   then throw TypeNotFoundError
  let oldParent = node.parent
  let newParent = ImageTable.findNewNode( oldParent )
  if newParent == null then
   call delayQueue.addNode( node )
   return
  let newnode = CreateNode( newtype, newParent )
  call CopyAttributes( node, newnode )
  call ImageTable.addImage( node, newnode )

ExecuteRule( rule, node )
  let matchResult = MatchRulePattern( rule, node )
  if matchResult == true then //Match succeeded
   if Eval( rule.condition ) == false then return //Can't apply
   let newtype = rule.mapsTo.RHS.type
   if rule.newParent == null then
    let oldParent = node.parent
    let newParent = ImageTable.findNewNode( oldParent )
   let newnode = CreateNode( newtype, newParent )
   call CopyAttributes( node, newnode )
   call Eval( rule.command )
   call ImageTable.addImage( node, newnode )
  else // Match failed, queue node
   call delayQueue.addNode( node )
```

Queuing and Delayed Rewrite. In certain cases, such as a migration rule that depends on a mapping for another node which has not yet been migrated, the migration for that node cannot be executed. But it may be possible to execute the migration after some other migration rules have been executed. We use a delayed rewrite approach to handle these cases, by queuing the nodes for which the migration cannot be immediately effected. The listing below describes this algorithm. After completing the first pass of the depth-first traversal, the queued nodes are processed by calling *ProcessQueue*. Nodes are removed from the queue (in FIFO order), and migration is attempted again. If *MigrateNode* fails, the node is added back at the end of the queue. If the length of the queue has changed after one pass, *ProcessQueue* is called again. The algorithm terminates when the queue is empty, or when a complete pass of the queue has not changed the queue.

Delayed rewrite algorithm

```
ProcessQueue( )
  let qLength = delayQueue.length
  if qLength == 0 return
  for index = 1 to qLength
   let node = delayQueue.removeTopNode
   call MigrateNode( node )
  if delayQueue.length < qLength then call ProcessQueue
```

5. RELATED WORK

Our work on model-migration has its origins in techniques for database schema evolution. More recently, though, even traditional programming language evolution has been shown to share many features with model migration. Drawing from experience in very large scale software evolution, [6] uses several examples to draw analogies between tradition programming language evolution and meta-model and model co-evolution. [3] also outlines parallels between meta-model and model co-evolution with several other research areas, including API versioning.

Using two industrial meta-models to analyze the types of common changes that occur during meta-model evolution, [9] gives a list of four major requirements that a model migration tool must fulfill in order to be considered effective: (1) Reuse of migration knowledge, (2) Expressive, custom migrations, (3) Modularity, and (4) Maintaining migration history. The first, reusing migration knowledge, is accomplished by the main MCL algorithm: meta-model independent changes are automatically deduced and migration code is automatically generated. Expressive, custom migrations are accomplished in MCL by (1) using the meta-models directly to describe the changes, and (2) allowing the user to write domain-specific code with a well-defined API. Our MCL tool also meets the last two requirements of [9]: MCL is modular in the sense that the specification of one migration rule does not affect other migration rules, and the history of the meta-model changes in persistent and available to migrate models at any point in time.

[5] performs model migration by first examining a difference model that records the evolution of the meta-model, and then producing ATL code that performs the model migration. Their tool uses the difference model to derive two model transformations in ATL: one for automatically resolvable changes, and one for unresolvable changes. MCL uses a difference model explicitly defined by the user, and uses its core algorithm to automatically deduce and resolve the breaking resolvable changes. Changes classified as breaking and unresolvable are also specified directly in the difference model, which makes dealing with unresolvable changes straightforward: the user defines a migration rule using a graphical notation that incorporates the two versions of the meta-model and uses a domain-specific C++ API for tasks such as querying and setting attribute values. In [5], the user

has to refine ATL transformation rules directly in order to deal with unresolvable changes.

[7] describes the benefits of using a comparison algorithm for automatically detecting the changes between two versions of a meta-model, but says they cannot use this approach because they use Ecore-based meta-models, which do not support unique identifiers, a feature needed by their approach. Rather than have the changes between meta-model versions defined explicitly by the user, they slightly modify the ChangeRecorder facility in the EMF tool set and use this to capture the changes as the user edits the meta-model. Their migration tool then generates a model migration in the Epsilon Transformation Language (ETL). In the case that there are meta-model changes other than renamings, user written code in ETL to facilitate these changes cannot currently be linked with the ETL code generated by their migration tool. In contrast to this, MCL allows the user to define complex migration rules with a straightforward graphical syntax, and then generates migration code to handle these rules and links it with the code produced by the main MCL algorithm.

[8] presents a language called COPE that allows a model migration to be decomposed into modular pieces. They note that because meta-model changes are often small, using endogenous model transformation techniques (i.e., the meta-models of the input and output models of the transformation are exactly the same) can be beneficial, even though the two meta-models are not identical in the general model migration problem. This use of endogenous techniques to provide a default migration rule for elements that do not change between meta-model versions is exactly what is done in the core MCL algorithm. However, in [8], the meta-model changes must be specified programmatically, as opposed to MCL, in which the meta-model changes are defined using a straightforward graphical syntax.

Rather than manually changing meta-models, the work in [13] proposes the use of QVT relations for evolving meta-models and raises the issue of combining this with a method for co-adapting models. While this is an interesting idea, our MCL language uses an explicit change language to describe meta-model changes rather than model transformations.

Although not focused on meta-model or model evolution, the work in [12] is similar to our approach. The authors perform the automatic generation of a semantic analysis model from a domain-specific visual language using a special "correspondence" model called a *meta-model triple*. The connections provided by the meta-model triple perform a similar role as the *MapsTo* and *WasMappedTo* links in MCL.

6. CONCLUSIONS

We have presented the Model Change Language (MCL), our language for specifying metamodel evolution and automatically generating the corresponding model migration. MCL requires the specification of only the evolved parts of a metamodel and automatically handles the persistent parts. The specification is done using the metamodels of the original and evolved language, which allows domain experts to use the same abstractions for specifying both metamodels and their evolution. Our implementation produces executable code to perform model migration from the evolution specification and has been integrated with our Model-Integrated Computing (MIC) metaprogrammable toolsuite and tested on a number DSML evolution examples of medium complexity. These test metamodels typically consisted of 50-100 elements, and the number of migration rules were on the order of 5-10. The examples were used in proof-of-concept demonstrations where savings in development effort were measured with promising results.

The model migration problem is an essential one for model-driven development and tooling, and there are several challenging problems remaining in this area. Efficiency of the migration code is of paramount importance, especially on large-scale models. The migration idioms that we have targeted were based on our past experience, but it appears that this should be an evolving set, to be extended and refined by other developers. Thus, a continuation of this work would need to address the problem of supporting such an extensible migration idiom set.

Acknowledgment. This work was sponsored by DARPA, under its Software Producibility Program. The views and conclusions presented are those of the authors and should not be interpreted as representing official policies or endorsements of DARPA or the US government.

7. REFERENCES

[1] A. Agrawal, T. Levendovszky, J. Sprinkle, F. Shi, and G. Karsai. Generative programming via graph transformations in the model-driven architecture. In *OOPSLA, 2002: Workshop on Generative Techniques in the Context of Model Driven Architecture*, 2002.

[2] D. Balasubramanian, C. vanBuskirk, G. Karsai, A. Narayanan, S. Neema, B. Ness, and F. Shi. Evolving paradigms and models in multi-paradigm modeling. Technical report, Institute for Software Integrated Systems, 2008.

[3] P. Bell. Automated transformation of statements within evolving domain specific languages. In *7th OOPSLA Workshop on Domain-Specific Modeling*, 2007.

[4] P. A. Bernstein and S. Melnik. Model Management 2.0: Manipulating Richer Mappings. In *SIGMOD 07*, 2007.

[5] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *12th International IEEE Enterprise Distributed Object Computing Conference, ECOC*, pages 222–231, 2008.

[6] J.-M. Favre. Meta-models and Models Co-Evolution in the 3D Software Space. In *Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA) at ICSM*, 2003.

[7] B. Gruschko, D. S. Kolovos, and R. F. Paige. Towards Synchronizing Models with Evolving Metamodels. In *Proceedings of the International Workshop on Model-Driven Software Evolution (MODSE)*, 2007.

[8] M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE: A Language for the Coupled Evolution of Metamodels and Models. In *MCCM Workshop at MoDELS*, 2009.

[9] M. Herrmannsdoerfer, S. Benz, and E. Jürgens.

Automatability of Coupled Evolution of Metamodels and Models in Practice. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS*, pages 645–659, 2008.

[10] MOF. Meta-Object Facility: Standards available from Object Management Group.

[11] J. Sprinkle and G. Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages and Computing*, 15(3-4):291 – 307, 2004. Domain-Specific Modeling with Visual Languages.

[12] H. Vangheluwe and J. de Lara. Automatic generation of model-to-model transformations from rule-based specifications of operational semantics. In *7th OOPSLA Workshop on Domain-Specific Modeling*, 2007.

[13] G. Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference*, pages 600–624, 2007.

Using Model-Based Testing for Testing Application Models in the Context of Domain-Specific Modelling

Janne Merilinna
VTT Technical Research Centre of Finland
P.O. Box 1000,
02044 Espoo, Finland
+358 442 788 501
janne.merilinna@vtt.fi

Olli-Pekka Puolitaival
VTT Technical Research Centre of Finland
P.O. Box 1100,
90571 Oulu, Finland
+358 400 606 293
olli-pekka.puolitaival@vtt.fi

ABSTRACT
Domain-Specific Modelling (DSM) has evidently increased productivity and quality in software development. Although productivity and quality gains are remarkable, the modelled applications still need to be tested prior to release. Although traditional testing approaches can be applied also in the context of DSM for testing generated applications, maintaining a comprehensive test suite for all developed applications is tedious. In this paper, the feasibility of utilizing Model-Based Testing (MBT) to generate a test suite for application models is studied. The MBT is seen as a prominent approach for automatically generating comprehensive test cases from models describing externally visible behaviour of a system under testing (SUT). We study the feasibility by developing a domain-specific modelling language and a code generator for a coffee machine laboratorial case and apply MBT to generate a test suite for the application models. The gathered experiences indicate that there are no technical obstacles but the feasibility of the testing approach in large-scale models and languages is still questionable.

Categories and Subject Descriptors
D.2.5 [**Testing and Debugging**]: Testing tools

General Terms
Experimentation and Verification

Keywords
Model-driven development; Verification; Test generation

1. INTRODUCTION
Quite often Domain-Specific Languages (DSL) and Domain-Specific Modelling Languages (DSML) are mentioned to attain 5-10 fold productivity gains compared to traditional software development practices [1]. The productivity increase is primarily caused by the Domain-Specific Modelling (DSM) basic architecture, i.e. DSML, a code generator and a domain-specific software framework. It is also often argued that utilization of DSM increases software quality by decreasing programs errors among other things [1].

While well-defined DSML promotes modelling of correctly defined applications and in this way decreases program errors, the ultimate reason for productivity gains and the decrease of program errors is achieved via automation. Code generators are responsible for systematically transforming application models to source code on the target platform. While code generators systematically transform application models to source code, they also systematically produce program errors. The difference between code generation and manually transforming software specification to implementation is when program errors produced in code generation look the same and there are many of the same kinds of errors, manually transforming software specifications to implementation results in various kinds of errors.

From a testing point of view, the difference is that it is always easier to pinpoint an error which emerges frequently and systematically compared to errors emerging in various parts and in various shapes in the source code. This has a direct impact on source code quality. While applications produced without code generation need to be corrected one at a time, all errors found and corrected in code generators contributes to the overall quality of the whole product family.

Although it should be easier to find errors produced by code generators, locating all errors in code generators is not a trivial process. It is highly unlikely that all paths in code generators are traversed every time the source code is generated therefore errors do not reveal themselves easily. Similar to traditional application testing, improving the level of quality of code generators requires extensive test suite. In the case of code generators the test suite is a set of application models similar to compilers in traditional software development where source code is an input to the compiler. Thus, to improve the quality of code generators requires an extensive set of application models. In [1], iterative and incremental DSM, the development approach is argued to produce DSMLs with code generators of good enough quality. In our earlier work [2] it was argued that a more systematic approach is required as the iterative and incremental development approach may not produce an extensive enough test suite for code generators. Therefore an approach to produce an extensive set of application models as a test suite is required.

In [2], we presented a concept for testing the whole DSM basic architecture. The approach consists of two phases: 1) generating application models from a metamodel with an approach of Model-Based Testing (MBT) [3], and 2) generating a test suite for generated application models with MBT. In this paper, we further elaborate the second phase and demonstrate the approach in a laboratorial case study. We gather the experiences in testing application models with MBT in a laboratory case for a coffee machine for which a DSML and Python source code generator were developed.

This paper is structured as follows. First, the principles of DSM and MBT are discussed to set a background and baseline for our work. Second, utilizing MBT as a means for testing application models is presented. Third, the application testing approach is demonstrated in a laboratory case involving a coffee machine. Discussion and conclusions close the paper.

2. BACKGROUND

To get an understanding of the DSM testing approach under scrutiny, the background of DSM basic architecture and MBT needs to be known. Next, in this section, the background of DSM and MBT are discussed.

2.1 Domain-Specific Modelling

Increasing productivity in software development is largely dependent on software reuse and automation. Often the work required to increase productivity follows the same pattern as Roberts et al. present in [4] when reusability is considered. First, a couple of example applications are developed according to traditional means. The applications of same product family share a set of components that can be reused in the product family. When the amount of reusable components increases, white-box and black-box frameworks begin to emerge. While the frameworks mature, the application development increasingly shifts from low-level programming to utilization of the developed framework. Ultimately, the development of applications may be about choosing different alternative features from a pre-defined feature tree. In the case where there is a considerable number of variation and neither feature-trees nor wizards can be utilized, domain-specific languages (DSL) and DSMLs start to emerge.

A DSM solution consists of three main parts, often described as DSM basic architecture [1]:

- A metamodel defines the syntax of a modelling language. In the case of DSMLs, a metamodel mirrors the problem-space by providing modelling elements found directly from the problem domain. In practice, the metamodel also includes elements and restrictions of the target platform.
- Code generators define the transformation rules on how to transform application models that are based on a metamodel to a source code representation.
- The software framework abstracts low-level details of the target platform and functions as a platform on which a code generator generates source code. Sometimes no framework is required and the generated code directly accesses the services and functions of the target platform.

Actual applications are modelled based on the model elements and constraints of the developed metamodel. The models can be transformed into source code or any given representation with generators.

2.2 Model-Based Testing

The MBT is a black-box software testing method in which test cases are automatically generated from a model describing the behaviour of a system under testing (SUT) [3]. The MBT consists of three phases, i.e. modelling, test generation and test execution.

In the modelling phase, behaviour of the SUT is modelled according to specifications of the SUT where functional requirements are the primary source for developing the MBT models [3]. The MBT being a black-box testing method, the MBT models are required to embody the externally visible behaviour of the SUT, i.e. input and output data of the SUT. The input data is used for executing the tests and output data for verifying the tests. The notation of the models can be graphical, textual or mixed where the notation varies from general purpose to domain-specific [5].

Test generation is based on model traversal where several test design algorithms are utilized for generating test cases from the model. For offline testing [6], i.e. generating a test suite first and then executing it, there are two categories of test design algorithms i.e. requirement-based criteria and coverage criteria [7]. Requirement-based criteria test design algorithms are based on model traversal algorithms that traverse the MBT models until all required parts of the model are visited. Coverage criteria test design algorithms aim to traverse the MBT models until a required coverage criteria is fulfilled. For online testing [6] walking test design algorithms are utilized [7]. In the walking test design algorithm approach, each subsequent test step is decided after executing a preceding test step. It must be noticed, that the coverage of the test suite can only be as extensive as the model describes, i.e. parts of the program behaviour not described in models are not tested.

In the test execution phase, the generated test suite is executed against the SUT. As the software implementation is developed from the same specification as the MBT models, two opinions regarding the behaviour exist. The difference between these opinions is seen as errors during test execution.

The main benefits of the MBT are the facilitation of test suite maintenance and the coverage of the test suites. The facilitation of the test suite is based on the supposition that only MBT models are required to be kept up-to-date when the SUT evolves and the test suite can always be updated via test generation. The increased coverage is based on sophisticated test design algorithms that are the result of long time research. [3]

3. TERMS OF UTILIZING MBT FOR TESTING APPLICATIONS

Testing of applications in the context of DSM can mean the following aspects when the utilized metamodel restricts modelling of incorrect application models:

- Does the modelled application satisfy its functional and quality requirements, i.e. is the application modelled correctly and according to specifications?
- Does the correctly modelled application model transform to a source code representation correctly?
- Does the correctly-modelled generated application function as modelled when executed?

In this paper, we concentrate on the two latter aspects. Thus we assume that the application models are always correctly defined and the reason for failure is always caused by either:

1. failure in code generation alone,
2. platform failure alone, or
3. a combination of the preceding.

Considering 1), we do not strive for white-box testing and we do not consider source code inspection but rather strive for black-box testing. Thus in this paper we solely concentrate on testing how the generated code integrated with the software platform function as a combination, i.e. black-box testing.

Such as presented in Section 2.2 the failure of a test is caused by an incorrectly implemented application or MBT model. Considering DSM, the application model is always correct (according to our terms) therefore the failure can be caused by F1) failure in code generation, F2) platform, F3) a combination of the preceding or F4) incorrectly defined MBT models.

If we apply MBT out of its initial purpose and generate the test suite directly from a formal specification (see illustration in Figure 1, see also [2]), i.e. an application model from which source code is also generated, we no longer compare during test execution whether the modelled application is performing according to the specifications as we take the application model as a fact. This in no way contradicts our initial terms.

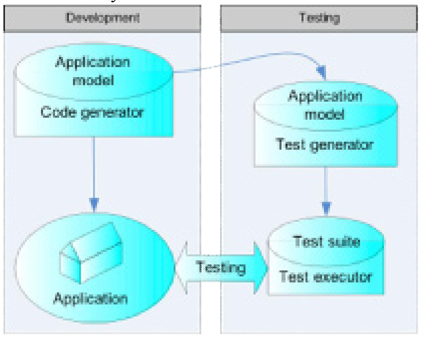


Figure 1. Using MBT for testing application models.

Now, as the same model is used as an input for code generation and test generation, we can rule out failure in F4 thus only F1-3 remain. F2 is also ruled out from discussion in this paper as it can be done using traditional testing approaches. Therefore a failure in code generation is the only thing remaining when the following terms are true:

- The application model is defined according to software specifications,
- The application model is correctly defined considering the utilized metamodel, and
- The test suite generated from application models is always correct.

4. A COFFEE MACHINE AS A LABORATORY CASE

To gather experiences and the technical limitations of utilizing the MBT in testing code generators in the context of DSM, a laboratory case example involving a coffee machine is utilized. The purpose of such machines is to take coffee orders as an input and deliver coffee as an output. There are also different kinds of machines where some are equipped with displays of various types and some machines require a different amount of money as an input whereas some make coffee for free. Nowadays there are also various blends of coffees available in addition to the basic combination of coffee and cream. Some of the most special coffees even have a very delicate preparation procedure thus producing a cup of coffee might be more than just the simplest procedure.

4.1 Tools for the Laboratory Case

As a language workbench for developing DSMLs, code generators and application models, MetaCase MetaEdit+[1] was chosen. MetaEdit+ includes tools to define DSMLs with GOPPRR (Graph-Object-Property-Port-Role-Relationship) metamodelling language and generators with MetaEdit+ Reporting Language (MERL) in addition to providing basic modelling facilities.

For the MBT of application models, there are two different approaches:

- develop test design algorithms within MetaEdit+ environment and generate a test suite by applying the developed generator, or
- take advantage of existing MBT tools.

The first approach requires implementing test design algorithms with MERL. The second approach requires exporting application models developed with MetaEdit+ to an external MBT tool. Exporting an application model requires implementing a model transformation specific to a metamodel with MERL. We chose the latter approach as developing a set of test design algorithms was anticipated as non-trivial and troublesome and we have had good experiences with MBT tools such as Conformiq Qtronic[2] (CQ), which was also chosen based on our evaluation presented in [7].

CQ expects the Qtronic Modelling Language (QML) as an input. QML is a variant of the Unified Modeling Language (UML) State Machine Diagram where as an action language a variant of Java is utilized. The action language is utilized for describing expected input and output values. From QML, CQ is able to generate test cases by applying a few coverage algorithms. CQ provides two pre-developed test scripters, which are used for generating Testing and Test Control Notation version 3 (TTCN-3) and Hypertext Markup Language (HTML). The TTCN-3 scripter produces a test suite described in TTCN-3 which can be used in test execution platforms. HTML scripter produces a UML Sequence Diagram as an illustration of the test cases. In addition, CQ provides a plug-in interface for the development of custom scripters.

4.2 Coffee Machine Modelling Language

For the coffee machine in question, Coffee Machine Modelling Language (CMML) was implemented with MetaEdit+. The CMML consists of two sub-languages where the first (see left hand side of Figure 2) is a User Interface Modelling Language (UIML) and the second language is a Coffee Making Process Modelling Language (CMPML) (see right hand side of Figure 2). The UIML enables testers to model the users interface (UI) of the coffee machine where the following aspects can be modelled: available sorts of coffee, the cost of a cup of coffee of a chosen sort, and textual information which is displayed to the user. The CMPML includes concepts for modelling e.g., heating a certain amount of water that is poured through a certain amount and blend of coffee to a cup of various sizes. Milk, cream, sugar etc. can be added when desired and foaming can be applied when required.

[1] www.metacase.com

[2] www.conformiq.com

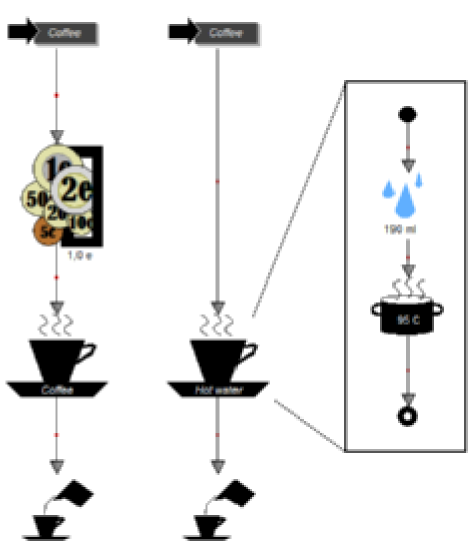


Figure 2. Simple coffee machine modelled with Coffee Machine Modelling Language.

For CMML, a Python source code generator was developed which produces complete code from models in the sense that there is no need to modify the generated code. The generated code enables simulation of the coffee machine behaviour in a desktop computer environment. The code generator is also able to produce a debug version of the modelled application in addition a stand-alone version. The debug version utilizes Simple Object Access Protocol (SOAP) Application Programming Interface (API) of MetaEdit+ to report a change of a state during the application execution back to the model where the application was generated. The change of a state is shown as a red rectangle highlighting the currently active state in the application models. This feature enables graphical debugging of the application.

4.3 Testing the Coffee Machine by Using MBT

The generated code follows an architectural division of UI, control logic (CL) and cooking machine interface (CMI). The UI is responsible for taking coffee orders and money as an input, and displaying information to the customer. The UI provides orders for CL to start making the ordered coffee. The CL is responsible for controlling the preparation process and it closely collaborates with CMI which simulates the physical hardware responsible for performing various preparation- related tasks. All components are implemented as Python threads to simulate concurrent processing and asynchronous events that are common to embedded devices. The preceding components have clearly defined interfaces to promote testability. No separate domain-specific framework exists because of the simplicity of the application domain.

In this laboratory case study, testing the behaviour of the CL is demonstrated. As discussed above, the CL has an interface to the UI and CMI. The provided interface of CL towards the UI consists of two kinds of signals i.e. pressing the coffee ordering button with the value of an ordered blend of drink, and the value of coin. The required interface towards the UI consists of text to the display signal. The required interface of CL towards CMI consists of the order signal with a value either to add water, add coffee, add milk, add cocoa, add cream, add sugar, warm water, and serve the coffee signal. The provided interface consists of a response signal with an ok/fail value as a parameter.

From an implementation perspective, i.e. after code generation, the signal exchange between the components in this example application is as follows when a customer orders a hot water product:

- CL receives a selected blend of a drink from UI.
- CL sends an add water order to CMI which immediately starts pouring the water to a heater.
- After CMI has finished pouring the water, it notifies the CL about the finished task.
- After receiving the pouring complete message from CMI, the CL notifies CMI to heat the water to 95 degrees.
- After CMI finishes heating the water, it notifies the CL.
- After receiving the heating complete message from CMI, the CL notifies CMI to serve the coffee.

4.3.1 Model Transformation

CQ expects QML as an input. As the metamodel of the CMML and the QML are different, a model transformation from CMML to QML is required. The model transformation can be divided into two main steps, i.e. transformation of the CMML objects and relationships into the QML state machine, and transformations of the information contained by CMML objects into QML input and output signals.

In the case of transforming the coffee machine application model to QML where CL is the SUT, the transformation is as follows. First, the QML state machine is initiated by generating QML Start, End and Idle states. The Start state has a transition to the Idle state. The Coffee Pressing Button objects (see the topmost entities in Figure 2) transform into QML transitions between the Idle and the Coffee decomposition states (see below). Serving the Coffee objects (see the lowest entity in Figure 2) transform into QML states and transitions to the End state. The Coin Input object transforms into a looping state which loops until the correct amount of coins is received. Transformation of Coffee objects depends on their decomposition. Transition to Coffee objects transform to a transition to the first object in a decomposition graph, and the transition from the Coffee object transforms to a transition leaving from the last object in a decomposition graph.

Objects of CMPML transform to QML states. Transitions entering to objects in CMPML transform to QML action transitions i.e. transitions that trigger an action represented by the connected object. Transitions leaving the CMPML objects transform to QML triggering transitions. Input and output values for QML action and triggering transitions are generated by considering values and types of CMML objects.

After the model transformation, the model is in the required format. The state machine part of QML is described in XML Metadata Interchange (XMI) format and action language for input and output transitions in QML. The result of the transformation does not include graphical presentation, however, for illustration purposes the transformed model can be manually visualized as shown in Figure 3.

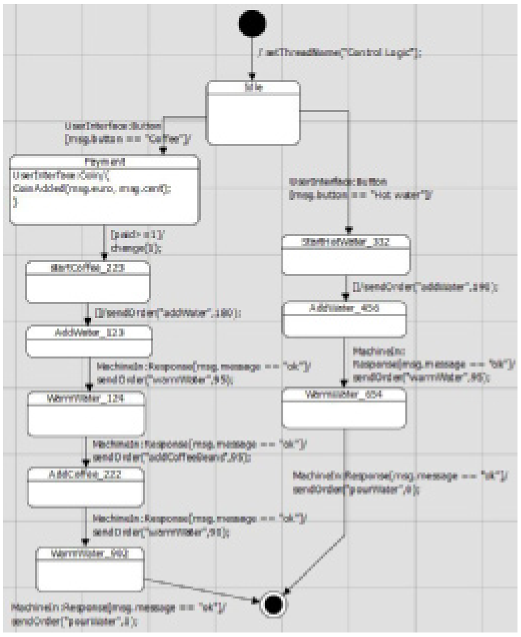

/ setThreadName("Control Logic");
Idle
UserInterface:Button
[msg.button == "Coffee"]/
Payment
UserInterface:Coin/{
CoinAdded(msg.euro, msg.cent);
}
[paid>=1]/
change(0);
StartCoffee_223
[]/sendOrder("addWater",180);
AddWater_123
MachineIn:Response[msg.message == "ok"]/
sendOrder("warmWater",95);
WarmWater_124
MachineIn:Response[msg.message == "ok"]/
sendOrder("addCoffeeBeans",95);
AddCoffee_222
MachineIn:Response[msg.message == "ok"]/
sendOrder("warmWater",95);
WarmWater_992
MachineIn:Response[msg.message == "ok"]/
sendOrder("pourWater",0);
UserInterface:Button
[msg.button == "Hot water"]/
StartHotWater_332
[]/sendOrder("addWater",190);
AddWater_456
MachineIn:
Response[msg.message == "ok"]/
sendOrder("warmWater",95);
WarmWater_654
MachineIn:Response[msg.message == "ok"]/
sendOrder("pourWater",0);

Figure 3. Qtronic model.

In Figure 3, the state machine part of the externally visible behaviour of the CL is presented. Input and output QML code is omitted in the figure for the sake of clarity. The left side of the figure represents ordering the coffee product and the right side represents ordering the hot water product.

4.3.2 Test Suite Generation

After the model transformation, a test engineer can choose which test design algorithms are to be utilized from the algorithms provided by the CQ. In this laboratorial example, transition and state coverage test design algorithms were chosen. The transition coverage test design algorithm generates a test suite in which each state of the model is visited at least once. The state coverage test design algorithm is similar to the transition coverage test design algorithm but visits all states. As an output format, HTML was chosen which is one the pre-made scripters provided by CQ. Now, CQ generates three test cases where one of the test cases is depicted in HTML format as in Figure 4.

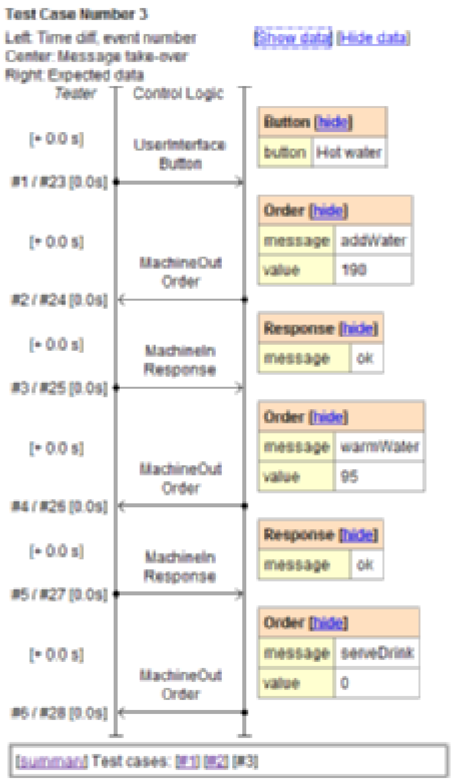


Figure 4. An illustration of a generated test case.

In Figure 4 a message exchange between the CL and the Tester is illustrated. As shown, CL receives *Button pressing event* with "*hot water*" parameter as an input from Tester, to which the CL reacts by sending *Order* with parameters "*addWater*" and "*190*" to Tester. The Tester reacts by sending *Response* with "*ok*". After that, the CL sends *Order* with parameters "*warmWater*" and "*95*" to Tester which again replies with "*ok*". After the CL receives the "*ok*" signal, it sends "*serveDrink*" with value "*0*" *Order* to the Tester which closes the test case. Now, if an error occurs when this test is executed, it is shown as a discrepancy of the expected output values.

5. DISCUSSION

For the feasibility study in utilizing MBT for testing applications developed with DSML for a coffee machine, MetaCase MetaEdit+ was chosen as a DSM environment whereas Conformiq Qtronic was chosen as an MBT tool. To enable the test generation, application models export from MetaEdit+ to QC was required. As the metamodels between MetaEdit+ and QC are different, a model transformation was required to transform CMML to QML, which is the format required by QC.

In the coffee machine laboratory case, the model transformation was trivial to implement as the mapping between CMML and the QML is straightforward. However, this might not be the case in real and perhaps more complex languages than the

CMML. As the MBT is completely dependent on the accuracy of the source model, even the slightest variation between the modelled behaviour of the application models and the MBT models ruins the test accuracy. In the case of CMML, model transformation was able to be validated by visualizing the MBT model but this again might not be possible when more complex languages are considered. It might be so that the chosen approach to utilize an external MBT tool might not be the perfect choice as the verification might just shift from testing the generated applications to testing the model transformations from the DSM to MBT environment. However, it must be noticed that such transformation has to be developed only once per language.

Another approach to utilize MBT is to replace an external MBT tool with test design algorithms developed directly in the DSM environment. This removes the need for model transformation but requires implementing the test design algorithms. Whereas by utilizing existing MBT tools that provide extensively-verified test design algorithms, now the test design algorithms have to be implemented for every language and a question about the quality of custom algorithm emerges. Currently, a trade-off when to utilize an external MBT tool compared to developing the test design algorithms by itself is a matter of debate and should be scrutinized before attempting to use MBT for testing a complete DSML, which is our ultimate target.

6. CONCLUSION

Industry cases constantly attain 5-10 fold productivity gains compared to traditional software development practices when DSMLs with full code generation is applied in software development. Not only are productivity gains witnessed but also quality is increased in the sense of decreased program errors. The increase of quality is partially explainable by well-developed modelling languages which prohibit the design of incorrect models but also because code generation has a remarkable impact on quality.

Although the quality increase is evident, software products cannot be released without proper testing without knowing that the modelling infrastructure, i.e. metamodels and code generators, is flawless. Iterative and incremental development of the modelling infrastructure is a state of the practice approach means to produce quality languages but still there is uncertainty about the quality without systematic and extensive testing of the whole infrastructure. Without such systematic and extensive testing methodology the resulting applications still need to be tested.

The contribution of this paper is a feasibility study of applying MBT to test the generated applications. While traditionally MBT models are developed from software specification parallel to implementation, we strive for applying the MBT to generate a test suite directly from domain-specific application models. In this way, the test suites are always up-to-date with the application models. In this paper, we demonstrated the utilization of MBT to generate test suites from application models in a laboratorial case study of a coffee machine for which DSML and Python code generator were developed. As a conclusion, the test generation seems to be technically feasible but it is still unknown if the approach is also feasible with more complex modelling languages.

7. REFERENCES

[1] Kelly, S. and Tolvanen, J-P. 2008. Domain-Specific Modeling: Enabling full code generation, John Wiley & Sons, ISBN 978-0-0470-03666, 427p.

[2] Merilinna, J., Puolitaival, O.-P. and Pärssinen, J. 2008. Towards Model-Based Testing of Domain-Specific Modelling Languages, The 8th OOPSLA Workshop on Domain-Specific Modeling, Nashville, TN, USA.

[3] Utting, M. and Legeard, B. 2006. Practical Model Based Testing: A Tools Approach, Morgan Kaufmann 1st ed., ISBN: 978-0123725011, 456p.

[4] Roberts, D., Johnson, R. 1996. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks, Proceedings of Pattern Languages of Programs Vol. 3 (1996).

[5] Hartman, A., Katara, M. and Olvovsky, S. 2006. Choosing a test modeling language: A survey, Haifa Verification Conference, pp. 204-218.

[6] Utting, M., Pretschner, A. and Legeard, B. 2006. A taxonomy of model-based testing, Working papers series. University of Waikato, Department of Computer Science, Hamilton, New Zealand, University of Waitako.

[7] Puolitaival, O.-P., Luo, M. and Kanstren, T. 2008. On the Properties and Selection of Model-Based Testing tool and Technique, 1st Workshop on Model-based Testing in Practice (MoTiP 2008), Berlin, Germany.

Right or Wrong? – Verification of Model Transformations using Colored Petri Nets*

M. Wimmer
TU Vienna
wimmer@big.tuwien.ac.at

G. Kappel
TU Vienna
gerti@big.tuwien.ac.at

A. Kusel
JKU Linz
kusel@bioinf.jku.at

W. Retschitzegger
University of Vienna
werner@bioinf.jku.at

J. Schoenboeck
TU Vienna
schoenboeck@bioinf.jku.at

W. Schwinger
JKU Linz
wieland@schwinger.at

ABSTRACT

Model-Driven Engineering (MDE) places models as first-class artifacts throughout the software lifecycle requiring the availability of proper transformation languages. Most of today's approaches use declarative rules to specify a mapping between source and target models which is then executed by a transformation engine. Transformation engines, however, most often hide the operational semantics of the mapping and operate on a considerable lower level of abstraction, thus hampering debugging. To tackle these limitations we propose a framework called TROPIC (Transformations on Petri Nets in Color) providing a DSL on top of Colored Petri Nets (CPNs) to specify, simulate, and formally verify model transformations. The formal underpinnings of CPNs enables simulation and verification of model transformations. By exploring the constructed state space of CPNs we show how predefined behavioral properties as well as custom state space functions can be applied for observing and tracking origins of errors during debugging.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification

Keywords

Model Transformation, CPN, Verification, Debugging

1. INTRODUCTION

MDE places models as first-class artifacts throughout the software life cycle, whereby model transformation languages play a vital role [5]. Several kinds of dedicated transformation languages are available (see [2] for an overview), the majority of them favoring declarative, rule based specifications to express mappings between source and target models, as is the case with the QVT Relations standard [1].

To execute the specified mapping, transformation engines are used, hiding the operational semantics and operating on a considerable lower level of abstraction. On the one hand this relieves transformation designers from burdens, like the necessity to specify a certain execution order. On the other hand, as transformation specifications grow larger, requiring numerous rules working together, this considerably hampers observing, tracking origins, and fixing of possible errors, being the main phases of debugging. As the correctness of the automatically generated target model fully depends on the correctness of the specified model transformation [13], formal underpinnings are required to enable verification of model transformations by proving certain properties like confluence and termination, to ease debugging [11].

To alleviate the above mentioned problems we propose TROPIC (TRansformations On Petri Nets In Color) [17, 18, 19], a framework providing declarative, reusable mapping operators based on a DSL on top of Colored Petri Nets (CPNs) [6] called Transformation Nets (TNs) to specify model transformations. TNs provide a homogenous representation of declarative mapping operators and their operational semantics, both in terms of CPN concepts. The formal underpinning of CPNs enables simulation of model transformations and exploration of the state space, which shows all possible firing sequences of a CPN. This allows applying generally accepted behavioral properties, characterizing the nature of a certain CPN, e.g., with respect to confluence or termination, as well as custom functions, e.g., to check if a certain target model can be created with the given transformation logic, during the observation and tracking origin phase of debugging.

The remainder of this paper is structured as follows. Section 2 introduces the main concepts of TROPIC. In Section 3, we show how properties of CPNs can be used to formally verify model transformations. Section 4 provides a taxonomy of possible transformation errors and related CPN properties, whereas Section 5 reports on lessons learned. Related work is discussed in Section 6, and finally, Section 7 provides an outlook on future work.

2. TROPIC IN A NUTSHELL

In the following, we shortly introduce TROPIC, a framework for model transformations applying CPN concepts. The use of Petri Nets in general enables a process oriented view on transformations. The abstraction from control flow prevalent in declarative transformation approaches is achieved as transitions can fire autonomously depending on the markings contained in the places only, although the statefulness of imperative approaches is preserved. CPNs, being a well-known class of Petri Nets, are most suited for model transformations, since every token carries a value of a certain type called token color, used to represent model elements accordingly. Thus CPNs provide a runtime model allowing transformation designers to gain an explicit, integrated rep-

*This work has been funded by the Austrian Science Fund (FWF) under grant P21374-N13.

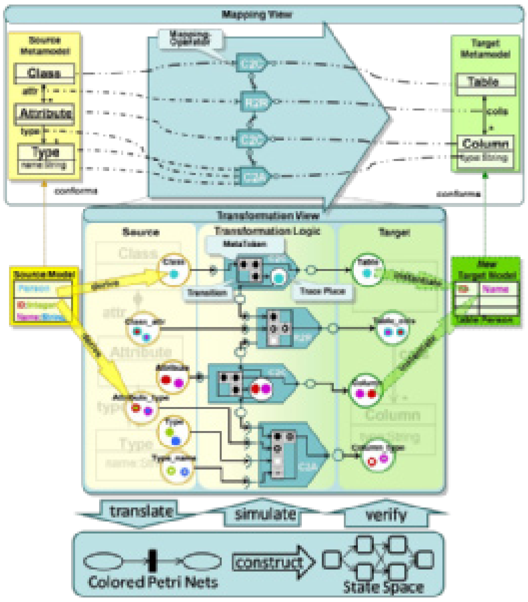


Figure 1: **Conceptual Architecture of TROPIC.**

resentation of the operational semantics of model transformations, which particularly favors debugging. To profit from these benefits of CPNs while hiding low-level details and circumventing restrictions thereof with respect to model transformations (cf. below), the TROPIC framework introduces a DSL for model transformations called Transformation Net (TN). TNs operate on two different levels of abstraction, providing a high-level mapping view and a more detailed transformation view (see Fig. 1).

Mapping View. The mapping view (upper part of Fig. 1) is used to declaratively define the correspondences between source (LHS) and target metamodel elements (RHS) using mapping operators (MOPs) encapsulating recurring transformation logic. MOPs are represented by means of *Hiearchical CPN concepts* [6], providing a packaging mechanism allowing a black-box view hiding the operational semantics of a transformation and a white-box view making the operational semantics explicitly (cf. below). In addition, a MOP's interface is only typed by classes, attributes and references being the main constituents of the Ecore meta-metamodel. This "weak typing" mechanism based on Ecore concepts allows to abstract from concrete metamodels, thus enabling reuse. For example in Fig. 1, showing a very simple transformation of classes to relations, the `C2C` MOP is used to simply map a class of the source model (`Class`) to a class in the target model (`Table`).

Transformation View. Every MOP of the mapping view requires a well defined operational semantics (i.e., the white-box view) in the form of some executable piece of transformation logic realized by an independent set of transitions and places. In particular, places are derived from elements of metamodels, creating a place for every class, attribute and reference thereof (see middle part of Fig. 1). Tokens are created from elements of models and then put into the according places. Finally, transitions represent the actual transformation logic. The existence of certain model elements (i.e., tokens) allows transitions to fire and thus stream these tokens from source places to target places to set up an unidirectional transformation. Tokens in target places finally represent instances of the target metamodel to be created and additional trace information is hold in terms of tokens within trace places. Note that the tokens in the target and trace places in Fig. 1 represent a successfully executed transformation. The LHS of a transtion representing the pre-conditions as well as the RHS depicting the post-conditions are visualized by means of color patterns (called `MetaTokens`). If a transition is enabled, the colors of the input tokens are bound to the input pattern. The production of output tokens is typically dependent on the matched input tokens. For instance, if a transition simply streams a certain token indicated by the same color pattern of `MetaTokens` on LHS and RHS, exact the same token is produced as output that was matched at the LHS. For example, the `C2C` operator is realized by such a transition taking a class (cf. arc from the place `Class` to LHS `MetaToken`) and producing an according table (cf. arc from RHS `MetaToken` to place `Table`) and additional trace information. For further details on TNs we refer to [17].

DSL on top of CPNs. TNs can be fully translated into existing CPN concepts, although, as already mentioned, adaptions have been made in order to better suit the domain of model transformations. These adaptions comprise, most importantly, a specific consumption behavior in order to deal with 1:n relationships and means to represent certain modeling concepts like multiplicity, ordered references and inheritance, accordingly.

First of all, the necessity of a specific consumption behavior is motivated in Fig. 2 by means of an example, depicting a simple TN generating a table for every class contained in a package. If token `P1` would be consumed (which is the default in CPNs), the transition could only fire once and the 1:n relationship between package and class would not be correctly resolved. Therefore, TNs provide a specific consumption behavior where tokens are not consumed per default. Rather the combinations of tokens fulfilling the precondition are held as trace information by every transition which allows firing for all possible combinations, which is typically desired in transformation scenarios. To represent the consumption behavior in CPNs we use an additional `History` place storing the already fired combinations in a sorted list together with an according guard condition, ensuring that the transition only fires if the current tokens of the precondition are not already contained in the history (see History Concept in Fig. 2b).

Regarding multiplicities of references, we provide *restrictions of places* in TNs represented by so called anti-places in CPNs. E.g., if the multiplicity of a reference in a target model is set to one, an anti-place holds exactly one token which is consumed if the reference is transformed, prohibiting repeated firings, cf. [9] for details. To cope with ordered references in a metamodel TNs introduce *ordered places* represented by lists in combination with anti-places in CPNs as they do not predefine a certain matching order for tokens. Inheritance between classes in a metamodel is depicted by means of *nested places* in TNs (place of superclass contains places of subclasses) and are represented by one place per class whereby places of superclasses additionally aggregate

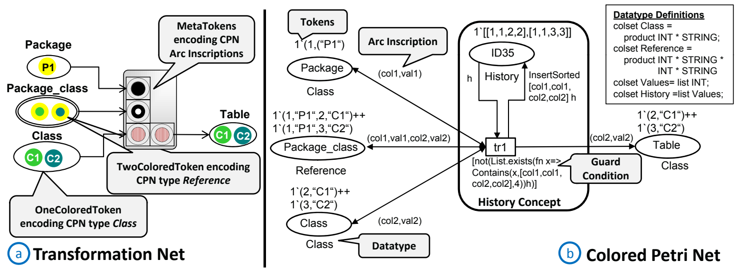


Figure 2: Translation of the Transformation Net Consumption Behavior to CPNs.

tokens of subclasses in CPNs. Additionally, to allow testing the absence of tokens, e.g., to create a class only if no link exists to a parent class (see transition c in Fig. 4a), TNs provide explicit concepts to represent *inhibitor arcs* hiding the CPN pattern presented in [9]. In contrast to metamodel specific concepts the translation of places, tokens and transitions itself is straightforward (places in TNs get converted to places with an according data type in CPNs, tokens in TNs remain tokens in CPNs, color patterns of transitions get converted to equivalent arc inscriptions, cf. Fig. 2).

3. PETRI NET-BASED VERIFICATION BY EXAMPLE

In the previous section we presented the foundations of TNs and their translation to CPNs which allows for the use of existing CPN execution engines to simulate TNs and, most importantly, the formal exploration of CPN properties. In the following subsections we present how properties of CPNs can be applied to verify model transformation specifications by means of an example.

3.1 UML2Relational Example

The example depicted in Fig. 3 and Fig. 4 is based on the Class2Relational case study[1], which became the de-facto standard example for model transformations. Due to reasons of brevity, only the most challenging part of this case study is described in this paper, namely how to represent inheritance hierarchies of classes within relational schemas following a simple one-table-per-hierarchy approach. As shown in Fig. 3 our example comprises three classes (cf. tokens in place `Class` in Fig 4a) whereby class `C2` inherits from class `C1` and class `C3` inherits from class `C2` (cf. tokens in place `Class_par` in Fig 4a). Therefore the desired output model should contain one `Table`, aggregating four `Columns` (all attributes of the three classes).

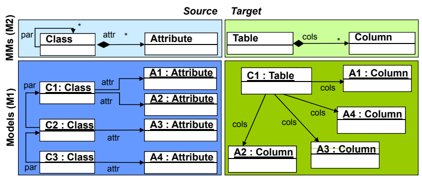


Figure 3: Metamodel and Model of UML2Relational Example.

At a first glance the generated target model in Fig. 4a seems to be correct, but on a closer look it can be detected, that a link from table `C1` to column `A4` is missing, compared to the desired target model depicted in Fig. 3. Even in this small example the error is hard to observe, but it is even more difficult to track the origin of the error. In the following we show how predefined formal properties of CPNs (cf. [10] for an overview) in combination with custom state space functions can ease these debugging phases using our current prototype.

3.2 Transformation Verification Prototype

As shown in Fig. 4, the created TN is translated to an according CPN, which allows the use of existing Petri Net execution engines, e.g., CPN Tools [2], enabling the simulation of model transformations. Although simulation can be used to get a first insight into the transformation specification, i.e., to investigate the operational semantics of the specified transformation, it is impossible to obtain a complete proof of *behavioral properties*, which require formal analysis methods of CPNs. Therefore the state space is constructed (see Fig. 4e), used on the hand to obtain predefined properties (see Fig. 4f), and on the other hand to analyze the transformation specification using custom functions, e.g., to check if a certain marking is reachable. The *Transformation Analyzer* component (see Fig. 4b) processes the analysis results, thereby verifying the specified transformation. Additionally to a source model and the specified transformation logic needed to calculate the state space, we assume that the expected target model is known, which is loaded by the *Transformation Analyzer* to derive the desired target markings in CPNs, which is then used for testing the transformation logic by applying custom state space functions.

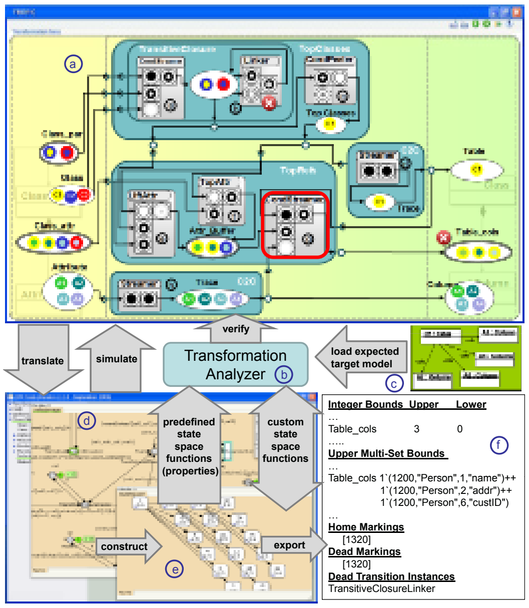
3.3 Verification of Model Transformations

In the following we show, how formal properties can be applied to detect errors in the transformation specification.

Model comparison using Boundedness Properties. Typically the first step in verifying the correctness of a transformation specification is to compare the target model generated by the transformation to the expected, manually created target model. To identify wrong or missing target elements in terms of tokens automatically, *Boundedness* properties (*Integer bounds and Multiset Bounds*) can be applied. In our example (cf. Fig. 4f), the upper integer bound of the `Table_cols` place is set to three whereas the desired target model requires four tokens, as every column has to belong

[1]http://sosym.dcs.kcl.ac.uk/events/mtip05

[2]http://wiki.daimi.au.dk/cpntools/cpntools.wiki


a
b
c
d
e
f
verify
translate
simulate
Transformation Analyzer
load expected target model
predefined state space functions (properties)
custom state space functions
construct
export

```
Integer Bounds  Upper     Lower
...
Table_cols          3         0
.....
Upper Multi-Set Bounds
...
Table_cols 1`(1200,"Person",1,"name")++
           1`(1200,"Person",2,"addr")++
           1`(1200,"Person",6,"custID")
...
Home Markings
    [1320]
Dead Markings
    [1320]
Dead Transition Instances
TransitiveClosureLinker
```

Figure 4: Transformation Verification Prototype showing the UML2Relational example

to a certain table. By inspecting the multiset bounds one recognizes that a link to the column A4 originating from an attribute of class C3 is missing. If such erroneous parts of the target model are detected, the owning target place (see error sign besides the `Table_cols` place in Fig. 4a) as well as the transitions that produce tokens in these places are highlighted in the TN. Unfortunately, numerous transitions are involved in creating the `Table_cols` link in our example, which hampers finding the actual origin of the error.

Transition Error Detection using Liveness Properties. Errors in the transformation specification occur if either a transition is specified incorrectly or the source model is incorrect. Both cases might lead to transitions which are never enabled during execution, so called *Dead Transition Instances* or *L0-Liveness*. The state space report in Fig. 4f shows that transition b in the TN is a *Dead Transition Instance*, which is therefore marked with an error sign. The intention of transition b in our example is to calculate the transitive closure, thus there should be an additional link from class `C` to class `A` as class `C` also inherits from class `A` (see Fig 3). On investigating the LHS of transition b in Fig. 4 we see that the inheritance hierarchy is faulty; the pattern specifies that class `X` (white color) is parent of class `Y` (black color) and class `Z` (gray color). To fix the color pattern we need to change outer and inner color of the second MetaToken; now class `X` (white color) is parent of class `Y` (black color), and `X` is again parent of class `Z` (gray color). After fixing the error, the state space can be constructed again and will not contain dead transitions anymore.

Termination and Confluence Verification using Dead and Home Markings. A transformation specification must always be able to terminate, thus the state space has to contain at least one *Dead Marking*. This is typically ensured by the history concept of TNs, which prevents firing for recurring combinations. Finally it has to be ensured that a dead marking is always reachable, meaning that a transformation specification is confluent, which can be checked by the *Home Marking* property requiring that a marking `M` can be reached from any other reachable marking.

The generated report in Fig. 4f shows that in our example a single *Home Marking* is available which is equal to the single *Dead Marking* (both identified by the marking 1320), meaning that the transformation specification always terminates. To achieve a correct transformation result, an equal *Home Marking* and *Dead Marking* is a necessary but not a sufficient condition, as it cannot be ensured that this marking represents the desired target model. By exploring the constructed state space using custom functions it is possible to detect if a certain marking is reachable with the specified transformation logic, i.e., the target marking derived from the desired target model. If it is, and the marking is equal to both, *Home Marking* and *Dead Marking*, it is ensured that the desired target model can be created with the specified transformation logic in any case.

4. CPN PROPERTIES FOR MODEL TRANSFORMATIONS

By applying and analyzing behavioral properties of CPNs in different case studies we tried to figure out which properties are useful in the context of model transformation verification and which kind of errors can be detected. The proposed taxonomy (see Fig. 5) investigates possible locations of errors, classifies typical model transformations errors and shows which properties are useful for their detection. The taxonomy extends our taxonomy presented in [8], focusing on how common QVT pitfalls can be detected in TROPIC.

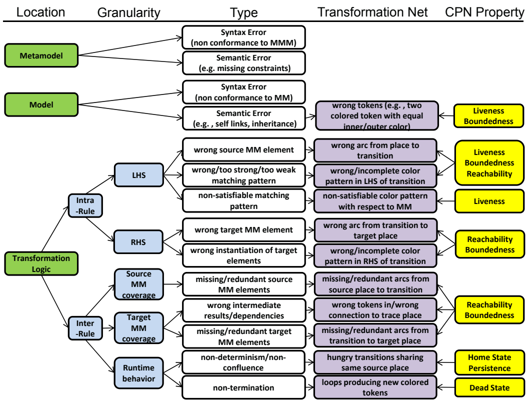

Location
Granularity
Type
Transformation Net
CPN Property
Metamodel
Model
Transformation Logic
Intra -Rule
Inter -Rule
LHS
RHS
Source MM coverage
Target MM coverage
Runtime behavior
Syntax Error (non conformance to MMM)
Semantic Error (e.g. missing constraints)
Syntax Error (non conformance to MM)
Semantic Error (e.g. , self links, inheritance)
wrong source MM element
wrong/too strong/too weak matching pattern
non-satisfiable matching pattern
wrong target MM element
wrong instantiation of target elements
missing/redundant source MM elements
wrong intermediate results/dependencies
missing/redundant target MM elements
non-determinism/non-confluence
non-termination
wrong tokens (e.g. , two colored token with equal inner/outer color)
wrong arc from place to transition
wrong/incomplete color pattern in LHS of transition
non-satisfiable color pattern with respect to MM
wrong arc from transition to target place
wrong/incomplete color pattern in RHS of transition
missing/redundant arcs from source place to transition
wrong tokens in/wrong connection to trace place
missing/redundant arcs from transition to target place
hungry transitions sharing same source place
loops producing new colored tokens
Liveness Boundedness
Liveness Boundedness Reachability
Liveness
Reachability Boundedness
Reachability Boundedness
Home State Persistence
Dead State

Figure 5: Taxonomy of Transformation Errors and CPN Properties.

During specification of model transformations there are three possible locations of errors, either in (i) the metamodel, (ii) the model, or (iii) the transformation logic. The detection of errors in the metamodel is in general out of scope of transformation languages. As we explicitly represent model elements—in contrast to other transformation languages—as tokens in TNs, semantic errors in the model can be detected by the liveness or boundedness property. For example, an incorrect source model (e.g., self links represented by two colored tokens with same inner and outer color) might lead to dead transition instances or an incorrect firing behavior of a transition and thus to an incorrect number of tokens in the target place.

Errors in the transformation logic itself can be divided into errors local to a single transition (*Intra-Rule Error*) or errors which can only be detected by examining the interrelations between several transitions (*Inter-Rule Error*). Intra-rule errors can be divided into errors occurring at the LHS or RHS of a transition. Common errors on both sides (e.g., a wrong matching pattern or a wrong instantiation of target models) can be detected by examining the boundedness properties in comparison to an expected target model or by custom state space functions checking if a certain marking is reachable. Due to the fact that these two properties can be applied in various scenarios we provide special tool support. If an expected target model is loaded, boundedness properties are automatically checked. Additionally, by selecting individual tokens of the desired target model (visualized in the TN editor by the *Transformation Analyzer* component), custom state space functions are created checking if the desired marking is reachable or not with the given transformation specification and the given source model. Finally, dead transition instances point out that the a given LHS specification of a transition cannot be fulfilled by the given source model.

Inter-rule errors occur if transitions depend on other, erroneous transitions or if we miss to cover the whole source or target metamodel. Although these errors can easily be detected by checking for source places that have no arc to any transition or target places which are not target of any transitions, it is also possible to apply boundedness and reachability properties to detect these kind of errors. To verify if several transitions in a model transformation specifications interact correctly, the confluence and the termination property can be applied. The creation of non-confluent transformation specifications in TNs might only occur if several transitions explicitly consume tokens from one place. If, in this case, the *Persistence* property is violated (the firing of one transition disables the firing of another one enabled before), which would lead to non-confluent model transformations, an error notification would be given to the transformation designer. As already stated and detailed in Section 5 the specific firing behavior of TNs ensures termination.

5. LESSONS LEARNED

This section presents lessons learned from the running example and thereby discusses key features of our approach.

History ensures termination. As mentioned above, TNs introduce a specific consumption behavior in that transitions do not consume the source tokens satisfying the precondition but hold them in a history. Thus, a transition can only fire once for a specific combination of input tokens prohibiting infinite loops, even for test arcs or cycles in the net. Only if a transition occurs in a cycle and if it produces new objects every time it fires, the history concept can not ensure termination. Such cycles, however, can be detected at design time and are automatically prevented for TNs. In contrast to model transformation languages based on graph grammars, where termination is undecidable in general [12], TNs ensure termination already at design time.

Visual syntax and live programming fosters debugging. TNs represent a visual formalism for defining model transformations which is, in combination with the exploration of formal properties, favorable for debugging purposes. This is not least since the flow of model elements undergoing certain transformations can be directly followed by observing the flow of tokens, allowing to detect undesired results easily. Another characteristic of TNs, that fosters debuggability, is live programming, i.e., some piece of transformation logic can be executed and thus tested immediately after definition without any further compilation step.

Concurreny in Petri Nets allows parallel execution of model transformations. As Petri Nets in general are especially suitable to specify concurrent operations, parallel execution of transformation logic is possible, thereby increasing efficiency of the transformation execution. In the UML2Relational example shown in Fig. 4a we can concurrently transform attributes to columns (cf. transition h) and calculate the transitive closure (cf. transition a). The properties *Home State* and *Dead Markings* can ensure confluence, even in case of parallel execution. The chosen representation of models by TNs let attributes as well as references become first-class citizens, resulting in a fine-grained decomposition of models allowing for extensive use of parallel execution.

State Space Explosion limits model size. A known problem of formal verification by Petri Nets is that the state space might become very large. Currently, the full occurrence graph is constructed to calculate properties leading to memory and performance problems for large source models and transformation specifications. Often a marking M has n concurrently enabled, different binding elements leading all to the same marking. Nevertheless, the enabled markings can be sorted in $n!$ ways, resulting in an explosion of the state space. As model transformations typically do not care about the order how certain elements are bound, Stubborn Sets [7] could be applied to reduce the state space nearly to half size, thus enhancing scalability of our approach.

6. RELATED WORK

The main objective of this paper is to provide formal verification methods for finding common transformation problems by employing CPNs. We consider two orthogonal threads of related work. First, we discuss other approaches which provide formal verification methods for model transformations. Second, we relate our proposed taxonomy to other error taxonomies in the domain of model transformations.

Formal verification of model transformations. Especially in the area of graph transformations some work has been conducted that uses Petri Nets to check formal properties of graph production rules. Thereby, the approach proposed in [15] translates individual graph rules into a Place/-Transition Net and checks for its termination. Another approach is described in [4], where the operational semantics of a visual language in the domain of production systems is described with graph transformations. The production system models as well as the graph transformations are transformed into Petri Nets in order to make use of the formal verification techniques for checking properties of the production system models. Varró presents in [14] a translation of graph transformation rules to transition systems (TS), serving as the mathematical specification formalism of various different model checkers to achieve formal verification of model transformation. Thereby, only the dynamic parts of the graph transformation systems are transformed to TS in order to reduce the state space.

These mentioned approaches only check for confluence and termination of the specified graph transformation rules, but compared to our approach, make no use of additional properties which might be helpful to e.g., point out the origin of an error. Additionally, these approaches are using Petri Nets only as a back-end for automatically analyzing properties of transformations, whereas we are using a DSL on top of CPNs as a front-end for fostering debuggability.

Error Taxonomies. In [13], a simple error taxonomy for model transformations is presented which is then used to automatically generate test cases for model transformations. A very similar approach is presented by Darabos et. al. in [3], focusing on common errors in graph transformation languages in general and on errors in the graph pattern matching phase in particular.

Both taxonomies are, however, rather general and describe possible errors in graph transformation specifications, only. Neither suggestions are presented how the findings of the generated test cases can be mapped back to the transformation specification in order to fix possible errors nor formal validation methods are presented.

7. FUTURE WORK

Up to now, we focused on small model transformation scenarios only, not least due to the state space explosion problem. The main disadvantage of the state space algorithms included in CPN Tools is that only full occurrence graphs can be constructed. The ASCoVeCO State space Analysis Platform (ASAP) [16], however, provides a tool to perform state space analysis on CPNs which tackles these shortcomings by allowing the specification of own, complex algorithms to construct and to explore the state space. We plan to integrate the ASAP tool into our prototype for evaluating different methods for their suitability in the domain of model transformations. Additionally, as the transformation logic by means of color patterns can easily become hard to comprehend when domain patterns grow larger, we plan to employ alternative visualization techniques, e.g., object diagrams or arc inscriptions known from CPNs.

8. REFERENCES

[1] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification, 2007.

[2] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3), 2006.

[3] A. Darabos, A. Pataricza, and D. Varró. Towards Testing the Implementation of Graph Transformations. *Electronic Notes in Theoretical Computer Science*, 211, April 2008.

[4] J. de Lara and H. Vangheluwe. Automating the Transformation-Based Analysis of Visual Languages. *Formal Aspects of Computing*, 21, Mai 2009.

[5] R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. *29th Int. Conf. on Software Engineering*, 2007.

[6] K. Jensen and L. M. Kristensen. *Coloured Petri Nets - Modeling and Validation of Concurrent Systems.* Springer, 2009.

[7] L. Kristensen and A. Valmari. Finding Stubborn Sets of Coloured Petri Nets without Unfolding. In *Poc. of Int. Conf. on Application and Theory of Petri Nets.* London, 1998.

[8] A. Kusel, W. Schwinger, M. Wimmer, and W. Retschitzegger. Common Pitfalls of Using QVT Relations - Graphical Debugging as Remedy. *Int. Workshop on UML and AADL @ ICECCS'09, Potsdam*, 2009.

[9] N. A. Mulyar and W. M. P. van der Aalst. *Patterns in Colored Petri Nets.* Beta, Research School for Operations Management and Logistics, 2005.

[10] T. Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4), 1989.

[11] F. Orjeas, E. Guerra, J. de Lara, and H. Ehrig. Correctness, completeness and termination of pattern-based model-to-model transformation. In *Proc. of 3rd Conf. on Algebra and Coalgebra in Computer Science*, Udine, 2009.

[12] D. Plump. Termination of graph rewriting is undecidable. *Fundamental Informatics*, 33(2), 1998.

[13] J. Uster, J. M. Küster, and M. A. el razik. Validation of Model Transformations - First Experiences using a White Box Approach. In *Proc. of MoDeVa'06 at MoDELS'06*, Genova, 2006.

[14] D. Varró. Automated Formal Verification of Visual Modeling Languages by Model Checking. *Journal of Software and Systems Modelling*, 3(2), 2003.

[15] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer. Termination Analysis of Model Transformations by Petri Nets. In *Proc. of 3rd ICGT*, Natal, 2006.

[16] M. Westergaard, S. Evangelista, and L. M. Kristensen. ASAP: An Extensible Platform for State Space Analysis. In *Proc. of 30th Int. Conf. on Application and Theory of Petri Nets and Other Models of Concurrency*, Paris, 2009.

[17] M. Wimmer, A. Kusel, T. Reiter, W. Retschitzegger, W. Schwinger, and G. Kappel. Lost in translation? transformation nets to the rescue! In *Proc. of 3rd Int. United Information Systems Conf. (UNISCON'09)*, Sydney, 2009.

[18] M. Wimmer, A. Kusel, J. Schoenboeck, G. Kappel, W. Retschitzegger, and W. Schwinger. Reviving QVT Relations: Model-based Debugging using Colored Petri Nets. In *MoDELS '09: Proceedings of the 12th international conference on Model Driven Engineering Languages and Systems*, Denver, 2009.

[19] M. Wimmer, A. Kusel, J. Schoenboeck, T. Reiter, W. Retschitzegger, and W. Schwinger. Let's Play the Token Game – Model Transformations Powered By Transformation Nets. In *Proc. of Int. Workshop on Petri Nets and Software Engineering*, Paris, 2009.

A Tooling Environment for Quality-Driven Domain-Specific Modelling

Janne Merilinna
VTT Technical Research Centre of Finland
P.O. Box 1000
02044 Espoo, Finland
janne.merilinna@vtt.fi

Tomi Räty
VTT Technical Research Centre of Finland
P.O. Box 1100
90571 Oulu, Finland
tomi.raty@vtt.fi

ABSTRACT

There is an increasing need for reducing costs and improving quality in software development. One of the means to reduce costs is to increase productivity by utilizing Domain-Specific Modelling (DSM). Industry cases consistently show a 5-10 fold increase in productivity when DSM is applied, in addition to a decrease of errors in generated code. In order to improve quality and especially desired quality attributes, e.g., performance and reliability, quality requirements must be considered in every development phase. Also a trace link from quality requirements definitions to implementation and tests has to be maintained to assure that the resulting application truly satisfies the requirements. As Model-Driven Development is heavily dependent on provided tool support, a tooling environment that enables quality-driven DSM would be useful. Thus in this paper, we study if MetaCase MetaEdit+ language workbench can be utilized as such by developing a code generator and a domain-specific modelling language for a laboratory case of stream-oriented computing systems. We found that the chosen environment is appropriate for an industrial application of quality-driven DSM.

Keywords

Model-Driven Development, quality attributes, traceability

1. INTRODUCTION

Whereas development costs must be abated in software development, at the same time customers demand products of ever higher quality. Today, it is not enough for software applications to satisfy demanding functional requirements. Rather, quality attributes such as the performance and reliability of an application also have to be planned, predicted, implemented and upon delivery it must attain its satisfactory and planned level of quality. Productivity must also be improved in software development to decrease development costs while achieving the desired quality.

One of the means to increase productivity is to apply modelling in software development. Model-Driven Development (MDD) treats models as first class design entities in which modelling is argued to provide a view to a complex problem and its solutions, an approach which is less risky, cheaper to develop, and easier to understand than the implementation of the actual target system [1]. In particular, the application of Domain-Specific Modelling (DSM) often results in a 5- to 10-fold increase in productivity in industrial cases in comparison to traditional practices [2].

To achieve products of desired quality, quality requirements have to be taken into account in software design and ultimately in an implementation. Much effort has been placed in developing methods and techniques that take quality requirements into account in software architecture development [3][4][5] and respectively to evaluate software quality from architectural models [6][7][8]. Whichever method is applied in architecture development where quality is of the importance, the following pieces must be employed for quality-driven development: 1) precise definitions of quality requirements, 2) a list of alternative design solutions to achieve such requirements, 3) linkage of the requirements and the design fragments that promote certain qualities, and 4) a method for utilizing such fragments in software design [9]. The preceding pieces must be accompanied with tracing of quality requirements. This is an activity of identifying requirements in the following work products through the entire development process [10]. The tracing improves all kinds of impact analysis from the measurable acceptance criteria of each quality requirement to the release of a software application [10].

To maintain a trace link all the way from quality requirements through architecture models to implementation, a link should not be broken at the model level. The models should also be automatically transformed into implementation to avoid an unnecessary phase of manually transforming models into source code. Considering the preceding statement, the subsequent requirements can be formulated for quality-driven DSM (QDSM): 1) quality requirements have to be explicitly expressed in models, 2) there must be a means to affect the quality attributes and their impact on quality should be observable in models, 3) there should be methods and techniques available to evaluate and test the models and the result of the evaluation and measurements should be presented in models to facilitate traceability of quality requirements. Finally, 4) full code generation from models has to be possible to enable testing of the models and to produce the release version of a modelled application.

Success of QDSM extensively lies in the provided tool support. Although there are tools to support quality requirements descriptions in architecture models [11], support for quality-driven development of architectures [12] and even support for architecture evaluation [11], we have not found a mature industry-ready integrated DSM tooling environment that demonstrates QDSM. Thus in this paper, we study if MetaCase MetaEdit+, which is a language workbench for developing code generators and domain-specific modelling languages (DSMLs) with a metamodelling approach, can be utilized as such an environment by developing a demonstration.

With MetaEdit+, we have developed a DSML and Python code generator which enables full code generation for simulating stream-oriented computing systems. The tool environment and the developed language provides: 1) a means to define quality requirements and to connect the requirements to corresponding model entities, 2) automated pattern recognition to provide a design rationale from the quality perspective, 3) measurement mechanisms for testing execution-time quality attributes [13] such as performance and reliability, 4) linkage between the measurement mechanisms and quality requirements to explicitly express if the quality requirements are satisfied, and 5) an optimization assistant that guides the modeller in achieving the desired qualities. We demonstrate the tool environment for modelling a system that initially does not satisfy its quality requirements, but with the help of the provided facilities of QDSM, is refined to fulfil the requirements.

This paper is structured as follows. First, requirements for QDSM including a state-of-the-art means for supporting quality-driven software development are discussed in Section 2. Second, a laboratory case including its modelling language and a code generator are introduced in Section 3. After that, in Section 4 the tool environment is demonstrated by transforming a model that does not satisfy its quality requirements into a system that completely satisfies the requirements. Discussion and conclusions close the paper.

2. Requirements for Quality-Driven Domain-Specific Modelling

The goal of QDSM is to entail in a single model the 1) quality requirements, 2) what has been done to satisfy the requirements, and 3) an evaluation and test results. By enabling this, tracing of quality requirements to implementation including test results is facilitated. Next, requirements and a state-of-the-art means for QDSM are more precisely discussed.

2.1 Expressing Quality Requirements

Quality requirements must explicitly be identified, divided and formalized to enable the designation of what parts of the application models are responsible for them and what are the means to validate the satisfaction of requirements. Considering MDD, quality requirements have to be declared not only in standalone requirements engineering tools but also in the modelling environment. Quality requirements have to be connectable to the modelling entities to maintain the explicit link between the requirements and the corresponding model entities.

There are several languages for the modelling of functional and quality requirements. These are evaluated in [14] and [15]. Also there are ontologies [16] and experimental visualization techniques [17] as well as existing tool support for requirements description, such as with IBM Rational RequisitePro and IBM Rational DOORS. Despite the format, it is of importance that the quality requirements should be defined in such way that their achievement can be verified, i.e. requirements should be measurable and they should include qualification requirements and acceptance criteria [10]. Such a template for describing quality requirements has been introduced by Ebert [10] for documenting quality goals.

2.2 Means to Affect the Quality Attributes

The utilized modelling language should have mechanisms to affect the quality attributes and there should be an enumeration of design approaches which enable to have an impact on quality attributes. It is also important that the impact of these mechanisms on the quality attributes should be made explicit to bridge the discrepancy between the quality requirements and the promoted quality attributes [18].

Patterns are considered as one of the means to express and affect the qualities of a software system. This argument is based on the definition of patterns. Alexander [19] defines patterns as "…a rule which establishes a relationship between context, a system of forces which arises in that context, and a configuration which allows these forces to resolve themselves in that context." Thus, patterns can also be seen as a solution for balancing forces related to qualities in a certain context. Currently, there is no extensive list of qualities that patterns promote. Nevertheless, some preliminary categorizations can be found from [3]. There are techniques, such as the goal-driven model transformation technique, that strives to provide a bridge between user requirements and design models [20] in which utilized patterns are based on scrutinizing the intent of patterns and dividing their intents into functional and quality parts. Then the most appropriate pattern is chosen for the situation at hand [21].

In addition to explicitly expressing the design rationale from the quality perspective, support for the modeller for the QDSM is recommended. Such support should include a model evaluation assistant that generates hints to optimize the system according to the quality requirements. Such hints can be based on e.g., architectural tactics [9] and patterns that promote the required quality requirements [3][12].

2.3 Evaluation and Testing

Models ultimately need to be evaluated and tested. There are a few software architecture evaluation methods that focus on certain quality attributes. The AEM method [7] concentrates on adaptability evaluation, whereas the IEE method [8] focuses on integrability and extensibility. There are also methods such as ATAM [6] that consider a set of quality attributes in the evaluation. While most of the evaluation methods are scenario-based or prediction methods, it is argued that quality can also be evaluated by inspecting what patterns are applied in models [12].

After evaluation, the models have to be tested by executing them to verify if the evaluation is tenable. Testing can only be performed for execution-time qualities [13] based on sheer definition. The generated implementation has to be monitored and measured from those parts in which quality requirements are connected to find out whether the execution-time quality requirements are satisfied. To support QDSM, the results of the tests need to be reported back to models. This enables the modeller to see measured values of quality attributes, and whether the quality requirements have been satisfied.

3. Domain-Specific Modelling Language for M-Net Laboratory Case

To demonstrate QDSM in MetaCase MetaEdit+, a laboratory case is utilized. The laboratory case is a stream-oriented computing system. For the laboratory case M-Net modelling language including a complete Python code generator was developed which enables complete code generation from domain-specific models. It must be noticed that the utilized laboratory case is only a laboratorial example of real stream-oriented computing system and is utilized only to simulate such a system.

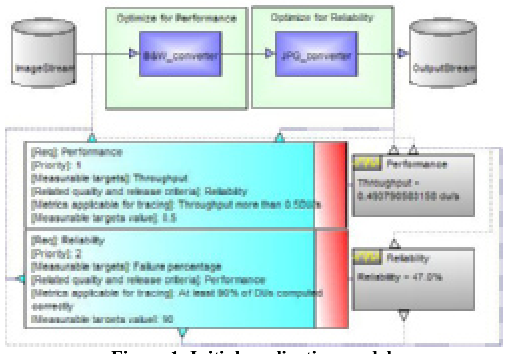
3.1 The Domain

Stream-oriented computing systems are characterized by parallel computing components that process potentially infinite sequences of data [22]. The purpose of such a system is to read data from a data source, manipulate the data and store or forward the computed data. Briefly, the system forms a pipes- and filters-based system which enables parallel processing of data. Similar systems are common, e.g., in video and image processing.

The domain of the laboratory case includes the following concepts (concepts in *italic*). *Filters* manipulate the input data and forward the filtered data to the next entity. The manipulation consumes time which is adjustable by the modeller. Computation can fail with a probability that the modeller can adjust to simulate e.g., insufficient resources during computation, program errors, or corrupted data units (DUs). If the computation fails, the modeller-definable penalty delay is endured. *Database* represents input and output pipes for the filter chain. *Switch* enables forwarding the data according to predefined principles. *Comparator* enables comparing the input data and based on predefined judgement policies, forward the input data to the next entities. *Pipes* connect various entities together.

The most relevant quality attributes are performance and reliability. Performance is the average throughput of DUs per second. Reliability is the average probability of computing and forwarding the data correctly. Reliability does have a direct impact on performance as when a *filter* fails to compute a DU, it suffers a penalty delay which has an impact on its performance.

Figure 1 represents an example of a stream-oriented computing system which consists of two *databases* and *filters*. The purpose of the application is to read a stream of colour bitmaps (from *ImageStream*) and transform the input into black and white (by *B&W_converter*) JPG images (by *JPG_converter*) and store the stream to a hard disc (*to OutputStream*).



Optimize for Performance
Optimize for Reliability
ImageStream
B&W_converter
JPG_converter
OutputStream
[Req]: Performance
[Priority]: 1
[Measurable targets]: Throughput
[Related quality and release criteria]: Reliability
[Metrics applicable for tracing]: Throughput more than 0.5DUs
[Measurable targets value]: 0.5
[Req]: Reliability
[Priority]: 2
[Measurable targets]: Failure percentage
[Related quality and release criteria]: Performance
[Metrics applicable for tracing]: At least 90% of DUs computed correctly
[Measurable targets value]: 90
Performance
Throughput = 0.490790583158 du/s
Reliability
Reliability = 47.0%

Figure 1. Initial application model.

3.2 Modelling Language and Code Generators for M-Net

For the laboratory case, an M-Net modelling language was developed with MetaEdit+. In addition to the basic metamodel and model manipulation mechanisms, MetaEdit+ also provides the possibility to alter the notation of the model entities during runtime, i.e. the notation of model entities may alter depending on, e.g., properties of the entities and/or relationships between the entities. This feature becomes useful when one wants to implement runtime model validation engines. Rules for altering the notation are defined with the same language as code generators and therefore the rules can be complex as necessary.

MetaEdit+ provides a domain-specific language called MERL for developing generators. Because generators have to be developed by self, the generated code will always be as desired which might not be the case with pre-made generators provided by tool vendors. This enables complete generation in the sense that the generated output is not required to be modified afterwards. MetaEdit+ also provides an Application Programming Interface (API) which is implemented as a Web Service interface with Simple Object Access Protocol (SOAP).

3.2.1 M-Net Modelling Language

The developed metamodel for M-Net includes all concepts existing in the domain. *Filters*, *databases*, *switches* and *comparators* are the main building blocks which are connectable with *pipes*. M-Net also includes additional concepts that promote QDSM. The quality requirements are described structurally in a *requirements* model entity which can be connected to the model to cover parts that are responsible for satisfying the requirement. The template for describing quality requirements is adapted from [10]. M-Net also includes *measurement mechanisms* that enable monitoring throughput and reliability of the modelled application. The *measurement mechanisms* are connected to *pipes* in the model, i.e. similar to using probes in electrical engineering, to measure parts of the model located between the probes. The o*ptimization assistant* model entity masks parts of the model and generates optimization hints for the modeller based on quality requirements.

3.2.2 Support for Quality-Driven Domain-Specific Modelling

The developed metamodel includes a pattern recognition mechanism which evaluates the model at modelling time and automatically recognizes if the modeller has successfully modelled any known predefined pattern that promotes certain quality attributes. To render the promoted quality attributes explicitly, entities in the pattern are automatically tagged by the pattern recognition feature with a text that informs what quality attributes the pattern promotes (see text on top of *filters* in Figure 2). The modeller can instantly experience if she has managed to model an application that manifests any patterns that promote certain quality attributes.

Requirements in M-Net language utilize MetaEdit+'s notation altering mechanisms to automatically inform the modeller if the requirement is satisfied (see tags on the right hand side of *requirement* entities located bottom left in Figure 1 indicating that the requirements are not satisfied and in Figure 2 where the requirements are satisfied thus there are no such tags). The automation is enabled if *requirement* entities are connected to corresponding *measurement mechanisms* that enable measuring throughput and reliability. Values for *measurement mechanisms* are reported to model during application runtime if an application is generated in debug mode (see details in Section 3.2.3.1).

The developed language also includes a model *optimization assistant* that can be utilized for guiding the optimization of the model according to the quality requirements (see two boxes that contain the *filters* in Figure 1). The *optimization assistant* considers only entities that it contains. This enables optimizing only the required parts of the model. The model optimization assistant utilizes a generator that traverses the entities it contains and generates optimization report on that basis.

3.2.3 The Generators

Two generators that transform the models to text were developed: 1) the Python source code generator and 2) the optimization report generator. The Python code generator is utilized for transforming the model into Python source code which is not required to be modified after code generation. The optimization report generator is utilized to generate textual hints for the modeller on how to optimize parts of the system according to the desired quality requirements.

3.2.3.1 Python Source Code Generator
Filters, *comparators*, *switches* and *databases* are generated as threads thus enabling parallel processing of data. The developed M-Net Python code generator also provides the option to produce additional code for the generated application that accesses MetaEdit+'s API to animate the entities in the models when the application is executed. When this option is selected, the preceding model entities are highlighted in models when they are active during the execution. This enables the modeller to see how the system functions in real-time and also at the model level.

Measurement mechanisms are generated with the application code only when the modeller chooses to generate a debug version of the application. The value for average throughput, i.e. performance, is disclosed in the execution of the application by counting DUs passing through parts of the application that the *measurements mechanism* monitor and by dividing the count by the time that elapsed to pass the DUs forward. The value for average reliability is calculated by counting the ratio between correctly computed DUs and corrupted DUs. Counted values are reported at real-time to the corresponding *measurement mechanisms* of the model by utilizing MetaEdit+'s API.

3.2.3.2 Optimization Report Generator
The optimization report generator finds all *optimization assistants* in the currently active diagram and generates optimization hints textually according to the desired quality requirements by considering the model entities it contains. The rules for such optimizations can be very complex in real-life domains but in this simple system the rules remain straightforward and simplified. An example of a trigger for performance optimization can be formulated in natural language as follows: *"Calculate average throughput of this entity. Find all entities forwarding data to this entity and compute the sum of the throughput. If the throughput sum is more than the throughput of this entity, performance optimization for this entity should be applied."* If a trigger for optimization is fired, a textual hint is generated that guides the modeller in refining the application design. An example hint for performance optimization can be as follows: *"B&W_converter can be optimized for "Performance" by: duplicating this element and adding switches before and after these entities. See pattern: performance optimization by duplication."* As presented, a hint always contains a link to a domain-specific pattern that promotes the desired quality attributes. The pattern catalogue is included with the modelling language as pre-made example models. This enables the modeller to discover what solutions are behind the optimization and provides additional information to the modeller for him/her to make the ultimate decision whether to apply the suggested optimization. The architectural knowledge is manually coded in the optimization hint generator.

4. Model Optimization According to Desired Qualities
QDSM is demonstrated by modelling an application that satisfies its functional requirements but not quality requirements. The first attempt to model an application is then transformed into a model that satisfies both the functional and quality requirements with the aid of the provided techniques for QDSM. The purpose of the example application is to convert a stream of bitmap images to black and white JPG images. The average performance requirement is >0.5 DUs per second. The average reliability requirement, i.e. correctness of the output, is >90%. In Figure 1, the first attempt to model such an application is presented.

4.1 First Iteration
ImageStream in Figure 1 contains DUs, i.e. colour bitmaps, which are required to be computed. The time to read data from *ImageStream* is 0.1s which is defined by the modeller to mimic real-life filters. The *B&W_converter* reads DUs into its buffer and immediately after receiving the first DU it starts computing the data. Time to compute the data of the *B&W_converter* is defined by the modeller to be 2s with 100% reliability. After computing each DU one at a time it forwards the DUs to a *JPG_converter* which stores the DUs into its buffer. The *JPG_converter* consumes 1s for each DU with an average of 50% reliability. If the *JPG_converter* fails to compute the DU correctly, it is defined by the modeller to suffer a penalty of 1s. It should be noted here that the values are artificial and are only for simulation purposes.

Performance characteristics of the initial version of the application are as follows. The throughput of the system can be calculated by considering the slowest part of the system. The throughput of the *B&W_converter* is 0.5DU/s as it takes 2s to compute each DU. Throughput of the *JPG_converter* can be calculated as follows. P(*JPG_converter*) = 1/(T(*JPG_converter*) + T(*JPG_converter*_penalty)*R(*JPG_converter*)) where P is throughput, T is time and R is reliability. Thus P(*JPG_converter*) is 1/(1s+0.5*1s) = 2/3DU/s ~= 0.66DU/s. Therefore the average throughput of the system is 0.5DU/s after the first DU is computed. Reliability of the system can be calculated as follows. R(system)=R(*B&W_converter*)*R(*JPG_converter*), where R is reliability. If R(*B&W_converter*) is 1 and R(*JPG_converter*) is 0.5 then R(system) is 0.5, i.e. 50% thus the system fails to meet its reliability requirement.

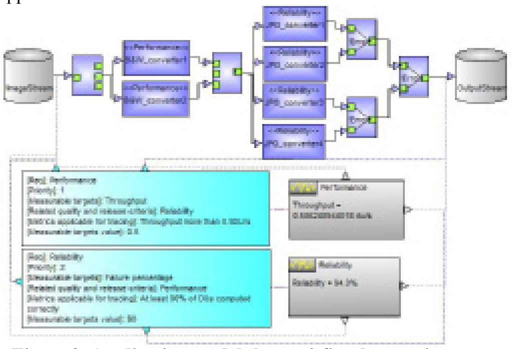
The performance characteristics can also be measured by connecting the *requirements* and *measurement mechanisms* to the model and by generating a debug version of the modelled application. The *requirements* and *measurement mechanisms* are connected to the application model to cover the whole computation part of the system such as Figure 1 illustrates. By doing so the modeller can see which requirements are meant to be satisfied and which entities are responsible for the requirements.

After code generation, the generated system can be executed. During run-time the system reports measurements back to the model and the requirements satisfaction indicators explicitly express if the requirements are satisfied. In this case, the system fails to meet both quality requirements (see tags on the right side of both *requirement* entities) as the average throughput is ~0.49DU/s and reliability 47%. The test was run by computing 100 DUs.

4.2 Second Iteration
The first attempt fails to satisfy the quality requirements resulting in the refinement of the application. The *B&W_converter* is the slowest part of the system and it has to be optimized for performance to increase the throughput of the application. *Optimization assistants* guide the optimization.

The *optimization assistant* enables the generation of optimization hints for the *filters* (see Section 3.2.3.2). Applying the hinted performance optimization pattern for the *B&W_converter* doubles the throughput of the optimized *filter* in an optimal case by enabling parallel processing of the data. The idea of this pattern is to forward the first input DU to the first *filter* which immediately starts processing the DU. The second DU is then forwarded to the second *filter* which starts computing the DU parallel to the first *filter*. In this way the *filters* receive every other DU and therefore halve the amount of DUs to be handled per *filter*. Thus, when the un-optimized *B&W_converter* produces 0.5DU/s, the throughput of the optimized *filter* combination is 1DU/s with 100% reliability. Now, the average throughput of the application is 0.66DU/s where the *JPG_converter* is the slowest part. The throughput requirement should be now satisfied.

Reliability of the application is still unsatisfactory therefore the *JPG_converter* has to be optimized for reliability. The *optimization assistant* produces the following hint for the *JPG_converter*: *"The JPG_converter can be optimized for "Reliability" by: duplicating this element and inputting the same data to both, and adding a comparator with the option "Error filter" after these elements. See pattern: reliability optimization by duplication."* The idea of this pattern is to compute the same data twice and forward the result to a *comparator* that forwards the corrupted DU only if both *filters* produce erroneous data. Otherwise the *comparator* forwards the first successfully computed DU. By applying this pattern the average reliability of the optimized *filter* increases, i.e. in this case reliability is increased to 75%. Applying this pattern twice results in an average of ~94% reliability. Figure 2 represents the optimized application model.



[Req]: Performance
[Priority]: 1
[Measurable target]: Throughput
[Related quality and release criteria]: Reliability
[Metrics applicable for tracing]: Throughput more than 0.5DU/s
[Measurable targets value]: 0.5

Throughput = [illegible] du/s

[Req]: Reliability
[Priority]: 2
[Measurable target]: Failure percentage
[Related quality and release criteria]: Performance
[Metrics applicable for tracing]: At least 90% of DUs computed correctly
[Measurable targets value]: 90

Reliability = 94.3%

Figure 2. Application model that satisfies the requirements.

In Figure 2, the application model that satisfies the both quality requirements is modelled. As can be seen, the developed pattern recognition engine automatically identifies the utilized patterns by tagging the corresponding *filters* with the promoted quality attributes. On top of the *B&W_converters* there is a <<Performance>> tag which ensures that these two *filters* participate in a pattern that promotes performance whereas in the *JPG_converters* there is <<Reliability>> tag. The tags should help the modeller to establish a design rationale for the application.

The calculated performance characteristics can be verified by generating an implementation from the model in debug mode and by executing the application. As *requirements* entities in Figure 2 does not have the tags on the right side, the application now satisfies the requirements. The average throughput measured by computing 100DUs is 0.58DU/s where reliability is 94%.

5. Discussion

We demonstrated QDSM with a laboratory-based case study of a stream-oriented computing system. For the system, an M-Net DSML and a code generator that enables full Python code generation from models was developed with MetaCase MetaEdit+. The most important means to support QDSM are based on 1) a quality requirements definition in models, 2) an automated model evaluation with pattern recognition, and 3) testing and reporting mechanisms.

Whereas a quality requirements definition technique is independent of the domain, model evaluation and testing can be considered domain-specific. In different domains, testing of execution-time quality requirements are largely dependent on what is measured and how. In this manner reusing the presented mechanisms between different domains is not possible. However, the concept remains. It is surprisingly useful to see the test results of an executed application also at the model-level. It is also useful to explicitly discover what quality requirements are satisfied. This enables to easily find out what requirements are satisfied and what parts of the model do not satisfy their quality requirements.

Design-time model evaluation for execution-time qualities was implemented in the laboratory case by identifying what patterns are utilized in the application model. As shown via the laboratory case, tentative design rationale can be obtained by utilizing pattern recognition. The evaluation could, however, also have included prediction of the performance characteristics as was done manually in the examples to provide more explicit values for quality attributes. Although evolution-time qualities such as modifiability and extensibility were not discussed in this paper, pattern recognition could also be utilized to provide some knowledge about the promoted evolution-time qualities. However, automation for quality evaluation except in the case of pattern recognition, which can provide tentative design rationale about the promoted qualities, might be a challenge for evolution-time qualities since scenario-based evaluation methods still need neural processing.

As discussed, it seems that pattern recognition can only provide tentative design rationale from quality perspective. Thus, it is questionable whether pattern recognition is sufficient for identifying the design rationale even in such a restricted area as DSM. In addition, different patterns promote different qualities and the utilized patterns might be overlapping in the application models. Therefore, finding out the design rationale by applying pattern recognition is not straightforward. In addition, sometimes it is not patterns that promote different qualities but more like functional blocks that are responsible for affecting a certain quality attribute. For instance, decreasing image resolution in image processing application certainly increases further image manipulation performance compared to utilizing high-resolution images. Decreasing image resolution might be a reason for optimizing the performance but sometimes the only reason for this is to satisfy a certain functional requirement.

As shown, by only identifying the utilized patterns it is not possible to discover the performance characteristics. Only a tentative design rationale can be obtained which, however, is still useful. Therefore to overcome the limitations with pattern recognition, next we will concentrate on manual approaches in describing design rationale by connecting requirements engineering side, where different techniques to affect the quality and their interrelated dependencies and impact to qualities are described, to the application development side. We already have developed a technique with tool support to provide measured performance characteristics from application models to requirements engineering side in order to ease the quality analysis in requirements engineering [23]. Next, we will connect the different design alternatives identified in requirements engineering to application development in a way that the impact of the utilized design alternatives to quality is automatically shown in application models. Thus, we rather strive for semiautomated approach than automated as it seems that humans cannot really be replaced by computers, yet.

6. Conclusion

There is a constant need for decreasing development costs in software development while at the same time increasing the quality of software applications. Increasing productivity can be achieved by utilizing MDD and especially DSM in software development. Nevertheless to increase the desired qualities of applications requires that not only the quality requirements must be considered at every development phase but that a continuous link from quality requirements to application design, testing and release must also be maintained. Maintaining such a link is crucial to reveal whether all the requirements set for software applications have been satisfied.

As the success of MDD extensively lies in the provided tool support, in this paper we demonstrated that there currently are mature integrated tooling environments, such as MetaCase MetaEdit+, that can be utilized as a platform for quality-driven DSM where the quality is traceable from quality requirements to application release. We demonstrated the tooling environment with a laboratory-conducted case study of a stream-oriented computing system.

7. ACKNOWLEDGMENTS

The work presented here has been developed within the MoSiS project ITEA 2 – ip06035. MoSiS is a project within the ITEA 2 – Eureka framework.

8. REFERENCES

[1] Selic, B. The Pragmatics of Model-Driven Development. IEEE Computer Society. IEEE Software, 2003, pp. 19-25.

[2] Kelly, S. and Tolvanen, J-P, Domain-Specific Modeling – Enabling full code generation, John Wiley & Sons, New Jersey, 2008, 427p., ISBN: 978-0-470-03666-2.

[3] Niemelä, E., Kalaoja, J. and Lago, P. 2005. Toward an architectural knowledge base for wireless service engineering, IEEE Transactions on Software Engineering, Vol. 31, No. 5, pp. 361-379. ISSN 0098-5589.

[4] Chung, L., Gross, D. and Yu, E., Architectural design to meet stakeholder requirements, The 1st Working IFIP Conference on Software Architecture, Kluwer Academic Publishers, San Antonio, TX, USA, 1999.

[5] Chung, L., Nixon, B.A., Yu, E. and Mylopoulus, J., Non-Functional Requirements in Software Engineering, Kluwer Academic Publishers, Boston, 2000.

[6] Kazman, R., Klein, M. and Clements, P., ATAM: Method for architecture evaluation, Carnegie Mellon University, Software Engineering Institute, Tech. Rep. CMU/SEI-2000-TR-004 ESC-TR-2000-004, 2000, 83 p.

[7] Tarvainen, P., Adaptability Evaluation at Software Architecture Level. The Open Software Engineering Journal, vol. 2, Bentham Science Publishers Ltd., 2008, pp. 1-30, ISSN: 1874-107X, http://www.bentham.org/open/tosej/openaccess2.htm

[8] Henttonen, K, Matinlassi, M., Niemelä, E., Kanstren, T. Integrability and Extensibility Evaluation from Software Architecture Models – A Case Study, 2007, Open Software Engineering. Vol. 1 No. 1, pp.1-20.

[9] Bachmann, F., Bass, L., Klein, M., Moving from quality attribute requirements to architectural decisions, In: Second International Software Requirements to Architectures, STRAW'03, 2003, Portland, USA.

[10] Ebert, C., Putting requirement management into praxis: dealing with nonfunctional requirements, Information & Software Technology 40(3): 175-185, 1998.

[11] Evesti, A. 2007 Quality-oriented software architecture development, VTT Publications 636, VTT, Espoo, 2007, 79p., URL: http://www.vtt.fi/inf/pdf/publications/2007/P636.pdf

[12] Merilinna, J., Niemelä, E., A stylebase as a tool for modelling of quality-driven software architecture, In Proceedings of the Estonian Academy of Sciences Engineering. Special issue on Programming Languages and Software Tools., vol. 11, No. 4, 2005, pp. 296–312.

[13] Matinlassi, M. and Niemelä, E., The Impact of Maintainability on Component-based Software Systems. In: 29th Euromicro Conference (EUROMICRO'03), Turkey, 2003, pp. 25-32.

[14] Carimo, R. A., Evaluation of UML Profile for Quality of Service from the User Perspective, Master's Thesis, Software Engineering, Thesis no: MSE-2007-03, August 2006.

[15] Etxeberria, L., Sagardui, G., Belategi, L., Modelling Variation in Quality Attributes, First International Workshop on Variability Modelling of Software-intensive Systems Limerick, Ireland — January 16–18, 2007.

[16] Savolainen, P., Niemelä, E., Savola, R., A Taxonomy of Information Security for Service-Centric Systems, Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications, 2007.

[17] Ernst N., Yu Y., Mylopoulos J., Visualizing non-functional requirements, In First International Workshop on Requirements Engineering Visualization (REV'06), Minneapolis, Minnesota, USA, 2006.

[18] Merilinna, J. and Räty, T., Bridging the Gap between the Quality Requirements and Implementation, The Fourth International Conference on Software Engineering Advances (ICSEA 2009), September 20-25, 2009 - Porto, Portugal, 6p.

[19] Alexander, C., The Timeless Way of Building, Oxford University Press, 1979.

[20] Lee, J. and Xue, N.L, Analyzing user requirements by use cases: A goal-driven approach. IEEE Software, 16 (4):92-101, July/August 1999.

[21] Fanjiang, Y-Y. and Kuo, J.Y., A Pattern-based Model Transformation Approach to Enhance Design Quality, In Proceedings of the 9th Joint Conference on Information Sciences (JCIS), 2006.

[22] StreamIt, Research overview page, URL: http://www.cag.lcs.mit.edu/streamit/shtml/research.shtml [Visited at 3.6.2009].

[23] Yrjönen, A. and Merilinna, J., Extending the NFR Framework with Measurable Non-Functional Requirements, 2nd International Workshop on Non-functional System Properties in Domain Specific Modeling Languages, Denver, Colorado, USA, Oct 4-9, 2009.

Towards a Generic Layout Composition Framework for Domain Specific Models

Jendrik Johannes[*]
Technische Universität Dresden
Institut für Software- und Multimediatechnik
D-01062, Dresden, Germany
jendrik.johannes@tu-dresden.de

Karsten Gaul
Technische Universität Dresden
Institut für Software- und Multimediatechnik
D-01062, Dresden, Germany
karsten.gaul@gmx.net

ABSTRACT

Domain Specific Models with graphical syntax play a big role in Model-Driven Software Development, as do model composition tools. Those tools however, often ignore or destroy layout information which is vital for graphical models. We believe that one reason for the insufficient support for layout information in model composition tools is the lack of generic solutions that are easy to adapt for new graphical modelling languages. Therefore, this paper proposes a language-independent framework for layout preservation and composition as an extension to existing model composition frameworks. We describe the single components of the framework and evaluate it in combination with the Reuseware Composition Framework for layout compositions in two different industrial used languages. We discuss the results of this evaluation and the next steps to be taken.

1. INTRODUCTION

In Model-Driven Software Development (MDSD) different graphical Domain Specific Models defined in different Domain Specific Modelling Languages (DSMLs) are used in combination. MDSD approaches promise high flexibility with regard to the DSMLs that are used and how these are combined. Using metamodelling tools, developers can create new DSMLs when required and integrate them into their MDSD process by defining model transformations and compositions. Different technologies are available for model transformation and composition which are language independent. That is, they can be used with any DSML that is defined by a metamodel they understand.

A drawback of such language-independent approaches is that they handle the semantic models, but rarely support the preservation and composition of layout information. This, however, is an important issue, because the outcome of a model composition is seldom the final system which is (like a compiled piece of code) processed by machines, but another model to be viewed and edited by developers. Thus, we argue that layout preservation and composition is crucial for the acceptance of MDSD. Currently, however, approaches that are easy to adapt for new DSMLs are missing.

This paper proposes such an approach for compositions of models within arbitrary graphical DSMLs (Section 2). It provides an implementation of the approach in an extensible framework that is based on the Eclipse Modeling Framework (EMF) [18]. Our framework can be used with arbitrary graphical DSMLs defined in EMF's metalanguage Ecore [18]. It can be connected to arbitrary EMF-based model composition engines that fulfill a number of properties we will discuss. One example of such an engine is the Reuseware Composition Framework [8] with which we evaluated our approach. We performed an evaluation of our framework with two different DSMLs (Section 3) used in industry. Afterwards, in Section 4, we discuss lessons learned and future extensions to broaden the scope of our framework. We look at related work in Section 5 and conclude in Section 6.

2. LAYOUT COMPOSITION

In this section we introduce the concepts behind our framework and, based on that, the different components of it. First (Section 2.1), we specify the scope of our work by defining criteria for the DSMLs and the model composition frameworks we support. Second (Section 2.2), we introduce the *Mental Map* concept on which we base our approach. Third (Section 2.3), we describe the different steps of our layout preservation and composition process and show variability within the different steps which can be implemented in individual components in our framework. Fourth (Section 2.4), we introduce the components we implemented.

2.1 Criteria for Supported DSMLs and Model Composition Frameworks

A DSML has to fulfill the following properties to work with our approach:
Requirement 1 The DSML has to have a graphical (diagrammatical) syntax.[1]
Requirement 2 The DSML has to be defined in Ecore.[2]

The following is required of a model composition framework to interoperate with our layout composition framework.
Requirement 3 The composition scripts for models have to have a graphical (diagrammatical) syntax.[1]
Requirement 4 The composition framework needs to be able to expose which item in a composition script refers to which input model.[1]

[*]This research has been co-funded by the European Commission in the 6th Framework Programme project MODELPLEX contract no. 034081 (`www.modelplex.org`).

[1]Section 4 discusses how these restrictions can be loosened
[2]This restriction applies if our implementation is reused directly. Conceptually, our framework can be ported to another modelling environment.

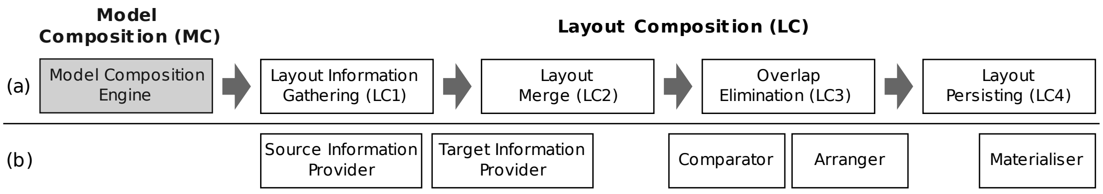


Figure 2: (a) Model and layout composition process and (b) Layout composition components of our framework

2.2 Mental Map

Naturally, when different diagrams are composed some adjustment of the layout is required because in many cases nodes of the former separated diagrams will overlap in the composed diagram. A naive solution would be to perform a complete relayout of the diagram using a layouting algorithm such as planarity [17] as shown in Figure 1.

This however, destroys the original neighborhood relationships between nodes. The literature calls these relationships the user's *Mental Map* [4] of the diagram. The importance of the Mental Map in MDSD is also stressed in [20]. One can think of the Mental Map as a road map, where the scale might vary, but the relations between elements do not. A user subconsciously creates his Mental Map of a diagram when arranging the icons in a certain way. Thus, when the layout is adjusted to eliminate overlaps the Mental Map should be preserved. There are three rules to be met in order to preserve the Mental Map [4]:

Goal 1: disjointness of nodes
Goal 2: keep the neighborhood relationship of the nodes
Goal 3: compact design

Naively applying layouting algorithms often violates one or more of these goals. In Figure 1, for example, the neighborhood relationship is not kept and, therefore, the result leaves the user disoriented.

2.3 Layout Composition Process

Figure 2a illustrates the model and layout composition process. The input to the process consists of one or more graphical models and one graphical composition script (Figure 3a). In the first step, the model composition engine—in our case Reuseware—interprets the composition script to perform the semantic model composition (MC). After that, our framework performs the layout composition (LC) with adjustment in four major steps. First (LC1), it collects the layout information from the input models and the composition script. Second (LC2), this information is merged in a Mental Map preserving fashion. For this, our framework needs to gather information from the underlying modelling and model composition frameworks (Requirement 4). Third (LC3), the merged layout data has to be adjusted to remove overlaps in a way that preserves the Mental Map (Goals 1–3). Fourth (LC4), the adjusted layout information has to be connected to the composed model, which again requires access to the underlying modelling technology.

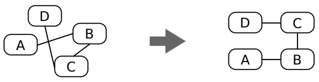


Figure 1: An application of the planarity algorithm that destroys the developer's Mental Map

In the first layout composition step (LC1) we collect all layout information. The collected information consists of (1) the layout information of each input diagram, (2) the layout information of the composition script and (3) the relation between nodes in the composition script and the input diagrams (Requirement 4).

The merging process (LC2) is steered by the layout of the composition script. The developer expects the composed model to be laid out according to his Mental Map of the composition script (cf. Figures 3a and 3b). Thus, using the information about how the nodes of the composition script relate to input diagrams, we move all the nodes of each single input diagram in correspondence to the node representing that diagram in the composition script. This is illustrated in Figure 3b where the element sets are arranged according to the composition script in Figure 3a. Because all nodes of one diagram are moved with the same vector, the Mental Map of the individual diagrams is preserved. Therefore, Goal 2 is reached. Since the positioning is based on the composition script, Goal 3 is also reached. The composed diagram however, may contain overlaps since the nodes representing models in the composition script are much smaller than the models themselves, which violates Goal 1.

To meet Goal 1, layout adjustment is performed in the next step (LC3). Here, we can make use of existing layout algorithms, where we treat all nodes that belong to one input model as a whole rather than adjusting each node individually (cf. adjustment from Figure 3b to Figure 3c). This is similar to the scaling of node clusters presented in [20].

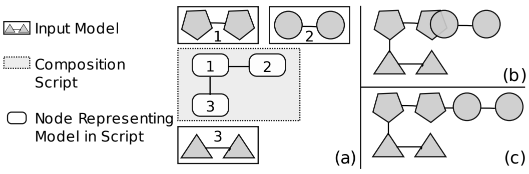


Figure 3: (a) Input of a model composition: 3 input models and 1 composition script (b) Composition result without overlay elimination (c) Composition result after the application of Horizontal Sorting

While a number of algorithms could be used (e.g., the ones discussed in [4]) we implement *Horizontal Sorting* [11] and *Uniform Scaling* [4] so far. Horizontal Sorting, as its name implies, starts at the left side of the diagram and moves overlapping fragments in x direction until they do not overlap anymore (Figure 3c). Uniform Scaling is based on the following equation: *(a+s*(x-a), b+s*(y-b))*. The point *(a,b)* is the center point and *(x,y)* is the location of the model element set that needs to be moved. The factor *s* is a scale factor used to define the distance the model elements are moved by. For automatisation purposes, *(a,b)* should not be chosen by the user—it can be computed from the merged diagram before adjustment (outcome of LC2, cf. Figure 3b).

In the last step (LC4) the computed layout information has to be materialised in the diagram of the composed model. Here, access for modifying this diagram has to be provided by the modelling technology that was used.

2.4 Framework Components

The previous section described the steps our layout composition framework performs. These steps can be implemented in individual components, which could be exchanged depending on specific demands of one composition. The different components are illustrated in Figure 2b. In the following, we give details of the functionality of these components and present what we have implemented so far.

Different component combinations can be used to achieve different results. We summarize possible combinations at the end of this section. In Section 3 we then evaluate what the benefits and drawbacks of certain combinations are.

2.4.1 Source Information Provider (LC1)

An input model consists of an arbitrary number of nodes and, for a user friendly layout algorithm that obeys the rules of the Mental Map, we have to know about the width and height of these nodes. More precisely, width and height of the bounding box of the whole input model is needed (x and y values are not important here). An *Information Provider* walks through the diagram structure and gathers the required data. The *Source Information Provider* depends on the layout format used for the input diagrams.

We implemented two *Source Information Providers* for two layout formats commonly used in EMF, which are the GMF Notation Model [7] and the TOPCASED Diagram Interchange format [19]. The GMF Notation Model is widely spread, because it is used by all DSMLs created with the GMF—a generative DSML development framework. TOPCASED is an alternative framework with similar functionality which currently offers a set of high quality UML editors. There are efforts to align both frameworks to obtain a common layout format in the EMF (possibly aligned with an upgraded version of the Diagram Interchange OMG standard [15]). In general, the Information Providers implemented by us already cover a huge amount of diagram syntaxes used in EMF. Our experience showed that an Information Provider can be implemented within hours.

2.4.2 Target Information Provider (LC1 and LC2)

Another *Information Provider* is required to obtain the layout information of the nodes in a composition script that represent input models. We call this a *Target Information Provider* because it determines the main structure of the composed diagram (cf. Figures 3a and 3b). It gathers the x and y values of the geometrical shapes that represent the input models in the graphical script. Height and width are not that important here. This Information Provider depends on the layout format used for composition scripts in the supported composition engine.

In Reuseware, composition scripts (called composition programs) are created in a graphical editor which was developed with GMF. Consequently, we implemented one *Target Information Provider* that depends on the GMF Notation Model for layout information and on Reuseware to obtain the relationship information (Requirement 4) between nodes in a composition script and input models.

2.4.3 Comparator (LC3)

A *Comparator* ensures that layout composition is performed in a deterministic order. It is required when the semantic model composition does not depend on a deterministic order, but the layout adjustment does.

We implemented one Comparator that sorts input models according to their x position in the composition script (i.e., the one given by the Target Information Provider). This is needed for the *Horizontal Sorting* algorithm but was also used for the *Uniform Scaling* algorithm to have a deterministic order (although any other deterministic Comparator could be used here).

2.4.4 Arranger (LC3)

An *Arranger* does the actual layout adjustment if overlaps exist. Therefore, it first checks for overlaps and decides if additional adjustment is required. An Arranger could do these steps repetitively, depending on the adjustment algorithm. That is, if after one adjustment overlaps do still exist, it can do another algorithm run.

As mentioned in Section 2.3, we implement *Horizontal Sorting* and *Uniform Scaling* as two different Arrangers. Depending on the concrete composition, both algorithms yield results of different quality, as we will discuss in Section 3.

2.4.5 Materialiser (LC4)

The last step is materializing the computed layout in an actual diagram. This is realized by a *Materialiser.*

Materialisers also have to be implemented for each layout format that should be supported. Thus, we implemented one for GMF and one for TOPCASED.

In the next section we evaluate our framework in combination with Reuseware using two graphical DSMLs, where one is utilising GMF and one TOPCASED as layout format. For that, different combinations of the mentioned component implementations are used. We call such a combination a layout composition *strategy.* The Source Information Provider and the Materialiser are always determined by the format used by the corresponding DSML. As Target Information Provider and Comparator we always use the only implementation we provided so far. The Arrangers however

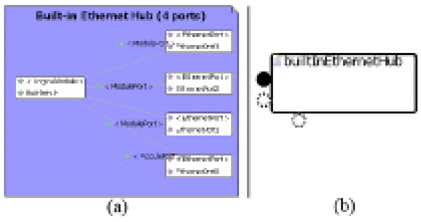


Figure 4: (a) A CIM model (b) The same model represented in a Reuseware composition script

can be varied (no Arranger, Horizontal Sorting, Uniform Scaling). We compare the different possible combinations and discuss the results.

3. EVALUATION

In this section we evaluate our layout composition framework on two different model compositions that were realized with the Reuseware Composition Framework in earlier works: [8] in Section 3.1 and [10] in Section 3.2. We apply different configurations of our layout composition framework and compare the results. Afterwards we discuss what we have achieved so far and what the next steps towards a generic layout composition framework are in Section 4.

3.1 Common Information Model DSML

The first model composition uses models from the telecommunications domain defined with a DSML that implements the Common Information Model (CIM) standard [3]. A metamodel defined in Ecore and a graphical GMF-based editor for the language were developed by Telefonica R&D and Xactium in the MODELPLEX research project [5]. In the following we concentrate on the layouting aspects of the model compositions. More details of the semantic model composition can be found in [10] and online[1].

Figure 4 shows (a) the CIM model `BuiltInEthernetHub` in the CIM GMF editor and (b) the representation of that input model in a composition script in Reuseware's composition script editor. The node in the composition script has different circles attached to it which are called *Ports* in Reuseware. Each Port points at a number of model elements in the input models that are modified during the model composition. For more details please consult [8] and the Reuseware website[2].

CIM models are composed in different stages, where each stage represents a different level of abstraction. The original input models (e.g., Figure 4a) developed with the mentioned GMF editor reside on Level 1. A composition script that composes these models defines a Level 2 composition. A composition script, that uses the results of Level 2 compositions as input models is located on Level 3 and so on.

The `BuiltInEthernetHub` (Figure 4a) is a model of Level 1. To compose the network model `EthernetIPInterface`, a composition script on Level 2 was created which is depicted

[1]http://reuseware.org/index.php/Abstract_CIM_DSLs
[2]http://reuseware.org

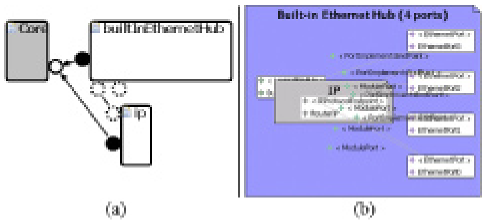


Figure 5: (a) Composition script for the `EthernetIP-Interface` model (b) Composed model

in Figure 5a. In addition to the `BuiltInEthernetHub` the script contains the input models `Core` and `IP`. The `Core` is an empty model into which CIM model elements are composed. Thus, it holds no graphical representation and layout information. The `IP` contains only one model element and consequently one graphical node.

We execute the composition defined in Figure 5a with three different layout composition strategies. Each strategy uses the Source Information Provider and Materialiser developed for GMF, the Target Information Provider that works for Reuseware's composition scripts and the Comparator. The first strategy applies no layout adjustment, the second uses Horizontal Sorting and the third Uniform Scaling.

No layout adjustment Figure 5b shows the diagram that results from the composition of Figure 5a without layout adjustment. We observe that the elements overlap (Goal 1) since the diagrams are bigger than the icons in the composition script (Figure 5a). This destroys the positioning in the developer's Mental Map (Goal 2) and only Goal 3 is reached.

Horizontal Sorting In Figure 6a Horizontal Sorting is used for layout adjustment. We observe, that the overlap has been removed. The overlapping nodes have been moved along the x-axis. While Goal 1 is reached here, Goal 2 is not completely satisfied. The `IP` model which is located below the `BuiltInEthernetHub` in Figure 5a is now located on the right of it.

Uniform Scaling In Figure 6b we utilise Uniform Scaling (with a scale factor s=2). Here, the mental map is well preserved and all three Goals are satisfied.

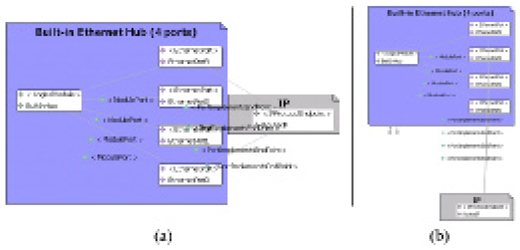


Figure 6: Adjusted layout: (a) Horizontal Sorting (b) Uniform Scaling

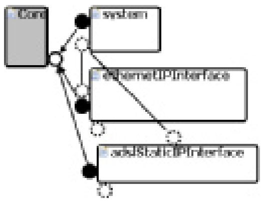


Figure 7: Level 3 composition script

We have seen that the layout adjustment is necessary even for a small model to avoid overlaps while preserving the Mental Map. While Horizontal Sorting performs worse than Uniform Scaling concerning the exactness of neighborhood relations, it yields a more compact design. Thus, it could still be an acceptable option here.

A more complex composition is shown in Figure 7. It is a Level 3 CIM abstraction that reuses results of earlier compositions on Level 2 which are the `EthernetIPInterface` from above as well as the models `ADSLStaticIPInterface` and `System`. We apply two different layout strategies using the two different algorithms to examine how they behave for larger models and how our framework behaves in a staged model composition.

Figure 8a (Horizontal Sorting) and Figure 8b (Uniform Scaling) show the different composition results. In principle, the same observations as above can be made, but the mentioned issues become more obvious. In the case of Horizontal Sorting, everything is aligned along the x-axis, while it was aligned along the y-axis in Figure 7. In the case of Uniform Scaling, the problem of less compact design increases. Although we used only a small scale factor (s=2), the diagram is getting relatively large. This is due to the fact that all element sets are moved uniformly in different directions, resulting into unused spaces between smaller element sets.

In Figures 8a and 8b we can also observe that a layout composed in an earlier step (i.e., the layout of `EthernetIPInterface` which is composed by Figure 5a) is not modfied anymore. Thus, different strategies can be applied at different stages of a composition. In Figures 8b, Horizontal Sorting was applied to compose `EthernetIPInterface` which keeps the overall layout more compact compared to the case where Uniform Scaling is applied everywhere (not shown).

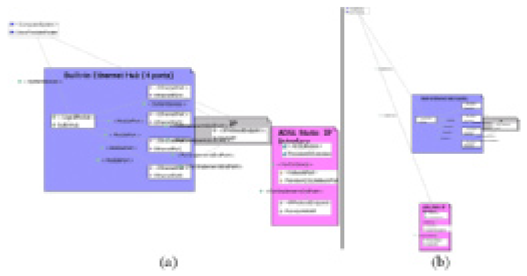


Figure 8: Composed Level 3 diagram: (a) Horizontal Sorting (b) Uniform Scaling

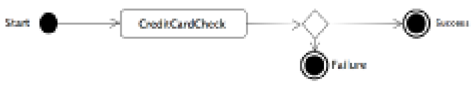


Figure 9: A business process extension modelled as UML activity in TOPCASED [19]

3.2 UML Activities for Business Processes

Business processes as presented in [6] can be modelled as UML activity models. A core process can be extended with new sub-processes by composing those models with Reuseware as we did in [8]. An example of a sub-process is shown in Figure 9. When it is composed into a larger core activity, the *initial* and *final* nodes (black circles) are replaced with other nodes in the core—integrating both activities.

We tested our layout composition framework with the models of [8]. This time, we had to use the TOPCASED specific components, since the diagrams were created with the TOPCASED UML editor. The adjustment worked in the same manner as for the CIM models confirming the results about strength and weaknesses of the algorithms and demonstrating that the framework can be used with other DSMLs.

4. NEXT STEPS

This section discusses the results of the last section and points out future work to improve our layout composition framework. We have seen throughout the evaluation that there is not one best strategy for layout adjustment. Which is the best strategy rather depends on many factors from the sizes of the input models up to the personal taste of the developer and how he uses the DSML at hand. A possibility is to make the developer aware of the different strategies and let him experiment with different ones—as we did in the evaluation. However, if many compositions are defined, this extra work of evaluating (and re-evaluating) all strategies each time a composition or one of its input models changes can become a tedious task. It should be possible to select strategies automatically based on further analysis of the input models or by allowing the developer to specify criteria for this selection. This requires analysis of a broader example space in the future.

Since we made certain assumptions about the models and the model compositions when we decided how to preserve the Mental Map, there are cases that are not so well supported by our framework at the moment. Consider the UML activity example (Figure 9). Here the nodes `Start`, `Success` and `Failure` are replaced by others during a composition. Currently, the layout information about these replaced nodes is always discarded. However, there are also examples where it seems to be more intuitive to position the replacing node at the position of the replaced node—for instance, if only one node is inserted and not a whole diagram. This however highly depends on the concrete kinds of compositions that are performed. If and how the best strategy can be determined automatically will have to be explored by evaluating different kinds of compositions. In addition, if a replacing node should take the position of the replaced node, the composition framework needs to reveal the relationship between such nodes (extension of Requirement 4).

Another thing we have not considered yet are diagrams that do not follow the simple node and edge paradigm but are more restrictive (e.g., UML sequence charts). In such cases, the layout adjustment possibilities are limited on the one hand, but might also not be necessary on the other hand (because a "good" layout is enforced by the nature of the graphical formalism). More investigations are required here.

A point that might hinder the combination with other model composition engines is the requirement for a graphical composition script (Requirement 3), since many such tools come with textual specification languages. In principle, such languages could also be handled by translating text positions (e.g., the order in which input models are referenced) into graphical layout information (by a specific Target Information Provider). Consequently, to support a textual language, a useful translation has to be found.

5. RELATED WORK

Many modelling tools do not pay proper attention to layout information today. Graphical modelling tools and frameworks such as GMF [7], TOPCASED [19], Rational Software Architect [9], Borland Together [2], MagicDraw [13] or Fujaba [12] offer facilities which apply layout algorithms to whole or partial diagrams. Despite the fact that these algorithms do not consider the Mental Map of the existing layout and often fail to produce viable results for large diagrams, the tools also do not offer facilities to preserve or transfer layouts from one diagram to another. MDSD process tools such as openArchitectureWare [14] or AndroMDA [1] completely ignore layout information when composing or transforming models.

Our work focuses on model compositions that are performed between models defined in one DSML. Another important discipline is model transformation between different DSMLs. Here, layout information is also seldom handled and also not considered in standardization efforts such as QVT [16]. Pilgrim et al. [20, 21] used trace links created during a model transformation to obtain layout information from the source diagram to layout the target diagram. They however do not discuss what the limitations for the model transformation are they support and only considered Uniform Scaling to remove overlaps so far.

6. CONCLUSION

We presented a generic layout composition framework to improve layout preservation in MDSD. The architecture of the framework and the components we implemented were introduced and utilised in several examples. These experiments showed that the provided solutions are a great improvement over current practice. They also showed, however, weaknesses and limitations of our work so far. Based on this, we identified challenges as a base for further work to improve the quality and genericity of the presented layout composition framework. In the future, we will tackle these challenges and perform more experiments on different DSMLs with distinct graphical syntaxes.

7. REFERENCES

[1] AndroMDA Development Team. AndroMDA. http://www.andromda.org/, 2009.

[2] Borland. Borland Together. http://www.borland.com/us/products/together/, 2009.

[3] Distributed Management Task Force Inc. (DMTF). Common Information Model Standards. http://www.dmtf.org/standards/cim/, 2008.

[4] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. *Research Report IIAS-RR-91-16E*, 1991.

[5] A. Evans, M. A. Fernández, and P. Mohagheghi. Experiences of Developing a Network Modeling Tool Using the Eclipse Environment. In *Proc. of ECMDA-FA'09*, volume 5562 of *LNCS*. Springer, 2009.

[6] M. Fritzsche, W. Gilani, C. Fritzsche, I. T. A. Spence, P. Kilpatrick, and T. J. Brown. Towards Utilizing Model-Driven Engineering of Composite Applications for Business Performance Analysis. In *Proc. ECMDA-FA'08*, volume 5095 of *LNCS*. Springer, 2008.

[7] GMF Development Team. Graphical Modeling Framework. http://www.eclipse.org/gmf/, 2009.

[8] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler. On Language-Independent Model Modularisation. In *Transactions on Aspect-Oriented Development*, LNCS. Springer, 2009. To Appear.

[9] IBM. Rational Software Architect. http://ibm.com/software/awdtools/architect/swarchitect/, 2009.

[10] J. Johannes, S. Zschaler, M. A. Fernández, A. Castillo, D. S. Kolovos, and R. F. Paige. Abstracting Complex Languages through Transformation and Composition. In *Proc. of MoDELS'09*, LNCS. Springer, 2009.

[11] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Research Report ISAS-RR-94-6E*, 1991.

[12] U. Nickel, J. Niere, and A. Zündorf. The FUJABA Environment. In *Proc. of ICSE'00*. IEEE, 2000.

[13] No Magic, Inc. MagicDraw. http://www.magicdraw.com/, 2009.

[14] oAW Development Team. openArchitectureWare. http://www.openarchitectureware.org/, 2009.

[15] Object Management Group. Diagram Interchange Specification, v1.0, 2006. http://www.omg.org/cgi-bin/doc?formal/06-04-04.

[16] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation, 2008. http://www.omg.org/cgi-bin/doc?formal/08-04-03.

[17] R. C. Read. A New Method for Drawing a Planar Graph Given the Cyclic Order of the Edges at Each Vertex. *Congressus Numerantium 56*, 1987.

[18] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *Eclipse Modeling Framework, 2nd Edition*. Pearson Education, 2008.

[19] TOPCASED Development Team. TOPCASED Environment. http://www.topcased.org, 2009.

[20] J. von Pilgrim. Mental Map and Model Driven Development. In *Proc. of LED'07*. EASST, 2007.

[21] J. von Pilgrim, B. Vanhooff, I. Schulz-Gerlach, and Y. Berbers. Constructing and Visualizing Transformation Chains. In *Proc. of ECMDA-FA'08*, volume 5095 of *LNCS*. Springer, 2008.

Model-Based Autosynthesis of Time-Triggered Buffers for Event-Based Middleware Systems [*]

Jonathan Sprinkle
University of Arizona
sprinkle@ECE.Arizona.Edu

Brandon Eames
Utah State University
beames@engineering.usu.edu

ABSTRACT

Application developers utilizing event-based middleware have sought to leverage domain-specific modeling for the advantages of intuitive specification, code synthesis, and support for design evolution, among other benefits. For cyber-physical systems, the use of event-based middleware may result, for some applications, in a need for additional time-based blocks that were not initially considered during system design. An advantage of domain-specific modeling is the ability to refactor an application to include time-triggered, event-based schedulers. In this paper we present an application in need of such modification, and discuss how these additional blocks must be synthesized in order to conform to the input/output constraints of the existing diagram.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Prog. Environments—*Integrated environments*

Keywords

Metamodeling, software synthesis, graph rewriting

1. INTRODUCTION

Cyber-physical systems involve algorithms and design techniques from the disciplines of control, real-time systems, robotics, software, communications, and many other application domains. Experience spanning multiple disciplines is required when developing these kinds of systems in order to manage the various subtleties of each domain, in addition to the subtleties of their integration.

In existing implementations of today's cyber-physical systems, designers commonly employ one of many discrete event-based computational models [25, 6, 11, 26]. In these models, components execute based on the exchange of tokens with other discrete event components. On execution, components acquire input tokens from their associated input queues, perform computation and submit their results to any consumers via output queues. A common event-based execution approach is realized through the application of event-based middleware, where (potentially) distributed components communicate through message passing in order to exchange data. The receipt of data typically triggers component execution.

Generally, components for such systems are developed by algorithm experts who understand well their computational behavior. Occasionally when a system is composed from components whose execution triggering rules are determined in an ad hoc manner by each developer, the behavior of the system is emergent in nature, as opposed to being engineered by design. A more principled system design could consider the semantics of the composition of the components (as discussed in [7]). However, such integration is difficult to enforce in a programming styleguide since the semantics are at a higher level.

In this paper, we discuss how to enforce a time-triggered, as opposed to purely event driven, behavior through the insertion of data buffers whose contents are read and released on the receipt of time-triggered tokens. We provide some examples of how existing domain-specific models of event-based middleware can be rewritten in order to produce new component graphs that now implement this kind of scalable behavior.

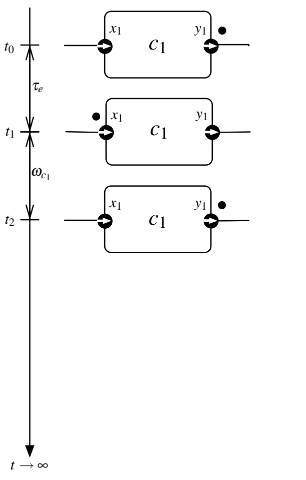
2. BACKGROUND

2.1 Middleware

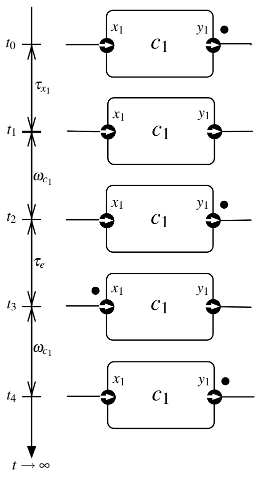
The growth in the number of middleware technologies, and their widespread adoption in large scale system design is a testament to their utility in mitigating low-level programming complexity in distributed system development. CORBA offers a middleware platform for supporting distributed computing. Real-Time Object Request Brokers (ORBS) [19] have been developed, e.g. TAO [22], which integrate operating system services and network protocols to offer predictable quality of service, including real-time or near real time response. TAO allows distributed applications to be specified as a set of interacting components. The middleware services manage data communication between components, including marshalling and demarshalling, allowing components to be written location-unaware. A variety of competing middleware technologies and platforms have been developed for supporting component-based distributed computing, including Ice [8], Enterprise Java Beans (EJB) [15], the Microsoft Component Object Model (COM) [3] and .NET framework [17], and Java RMI [24].

A key goal with middleware is the development of Quality of Service guarantees (QoS) for supporting application execution. Certain metrics are critical to distributed application development, e.g. bandwidth, latency and jitter. Different studies have been conducted to evaluate and improve the

[*]This work supported by the Air Force Research Labs, under award #FA8750-08-1-0024, titled "MultiCore Hardware Experiments in Software Producibility."



(a) Event-Based Execution, $\tau_e \leq \tau_{x_1}$



(b) Timeout Execution, $\tau_e > \tau_{x_1}$

Figure 1: Alternative execution of a component with timeout on a blocking read. (a) A token is received before, or at, the timeout of the blocking read. (b) No token is received by time t_1, so a timeout event occurs, and the output token is based on the previous input to the component. Receipt of later events occurs as in (a).

ability of middleware to offer predictable quality of service [21, 20]

2.2 Publish/Subscribe Methods

Publish/Subscribe is a common model of communication used for data interchange between components. A component c_1 will subscribe to another component's production of a particular data value, using services offered by the middleware. Then, the middleware will ensure that all subscribers receive the value once it is produced. This can be considered to be closely related to the producer/consumer model, however the producer/consumer model typically only involves a single consumer for a given producer. The publish/subscribe model offers explicit support for a controlled broadcast communication.

Several middleware technologies support variants of the publish/subscribe communication model, including Ice [8] and DDS [16]. Further, different approaches for implementing the model exist, including having the consumer task poll the producer for data availability, or having the producer perform a remote invocation on the consumer when data is available.

2.3 Time-Triggered Methods

Different techniques and infrastructures have been developed to support distributed/embedded computing which executes cognizant of time. Giotto [9, 10] offers a framework for capturing and executing time triggered software. Applications are captured as a set of interacting tasks, each of which is assigned to a mode with an associated execution period. Scheduling and inter-task communications are managed by Giotto's runtime infrastructure, called the E-machine.

Farcas et al. [5] have developed a component model to support the development of distributed real time systems. Components are decoupled from the platform and from each other, both temporally and functionally. The component model is based on the logical execution time (LET) model introduced with Giotto. Auerbach et al. [1] describe a Java-based approach for scheduling real-time control tasks on a quad-rotor model helicopter by making use of exotasks and the LET model to guarantee deterministic execution.

The Time Triggered Architecture (TTA) [12] and Time Triggered Protocol (TTP) [13] have been developed to support the time-based execution of components in hard real-time systems. Components are allowed to access shared communication hardware based only on the clock, rather than need. Design time scheduling determines *a priori* the allocation of access windows to shared busses. The goal of time triggered execution is to isolate the impact of component or subsystem failures on the system as a whole.

3. DOMAIN SEMANTICS

The domain of event-driven component-based systems provides significant freedom in the implementation of execution semantics. This freedom, in part, motivates the need to constrain execution across implementation platforms.

Our purpose in this section is to underscore the *semantic* reasons for which components specified in our language could execute differently on machines with different network latencies, or processing power. This understanding motivates why we undertake this transformation process, and additionally provides the semantics necessary for the components which we synthesize using the model transformations in the next section. We provide here several examples that demonstrate the subtle behavior anomalies that are a result of a purely event-driven model of computation. We continue with a concise description of the execution semantics of a time trigger component, which will be integrated into the system to eliminate these anomalies.

3.1 Single Input/Single Output

Consider a component, c_1, with a single input, x_1, and single output, y_1. We use subscripts to denote the component, input, and output index, respectively, in order to provide consistent labeling for multi-component, multi-input/output systems that we will describe in future sections. The value y_1 is obtained as the output of the functional behavior of the component, which may also be written in difference equation form as $y_1(k+1) = f(x_1(k))$, demonstrating the discrete notion of the software component, and our ability to encode y, x as signals in time (specifically, discrete time)[1]. The execution time of c_1 is not necessarily a constant, though we represent it by the variable ω_{c_1} for simplicity of specification, and leave open the potential that ω_{c_1} may be a random variable. In hard real-time systems ω_{c_1} (the execution time of the function) can be determined through worst-case execution time (WCET) analysis [18], and perhaps even be validated at the hardware level [4]. This component also has associated with it a time, τ_{x_1}, which is a timeout constant.

An example execution based on events received is shown in Figure 1a. At t_0, the logical beginning of this cycle, the system has just produced an output on y_1. At time $t_1 = t_0 + \tau_e$, an event occurs where a token is received on the input port, x_1. The component executes, and at time $t = t_2 = t_1 + \omega_{c_1}$ a token is produced on the output port, y_1, i.e., $y_1(t_2) = f(x_1(t_1))$. This execution is valid for any system execution where $\tau_e \leq \tau_{x_1}$, or where the component does not utilize a timeout.

In order to see how tokens flow in an execution where timeout is a factor, examine Figure 1b. At t_0, the system has just produced some token. At some time, $t_1 = t_0 + \tau_{x_1}$, a timeout event occurs, so the output port produces another token, namely $y_1(t_2) = y_1(t_0)$. That is to say, the output at $t_0 + \tau_{x_1} + \omega_{c_1}$ is equivalent to that produced at t_0. This equivalency ignores metadata such as send/receive timestamps, packet size, etc. Another remark is that the duplicative behavior of this component need not be the only behavior in terms of timeout. Many components may opt to produce a special timeout token, or no token at all, in the case of timeout. This is also a valid option, though in our case we aim to address systems that integrate tightly with physical systems, and inaction may be inappropriate in this domain.

An activity diagram that considers each of the cases presented in Figure 1a and Figure 1b is shown in Figure 2. In this diagram, the mutually exclusive case of receiving data, or timing out, is clearly shown. However, designs such as this (if already fielded) will satisfy these observations: (1) algorithms to discard "stale" data must already exist; and (2) timeouts, τ_{x_i}, will be appropriate for the execution time ω_{c_i} of this component, as well as the frequency of execution for all providing components.

[1]Of course, the internal state of an object can affect this outcome, but externally the interface is as presented.

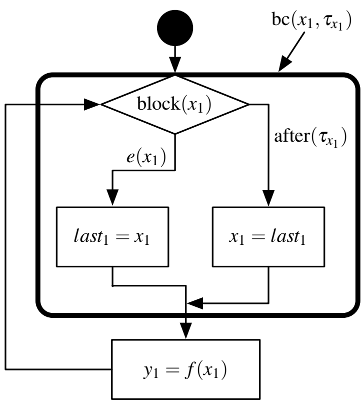


Figure 2: Activity Diagram that incorporates behaviors seen in Figure 1a and Figure 1b. The notation $e(x_1)$ indicates that an event was received indicating new data on input x_1; these events are cached if not being blocked upon, but the number of data values cached is application dependent.

For brevity, we mention that the approach, and issues, for SISO systems can be generalized for MIMO systems. Due to the scope of the workshop, we leave this discussion to future papers in the domain, rather than the language elements of our work.

3.2 Trigger Generator

We provide the semantics for a particular component called a trigger generator. This component produces, at specific times or rates, a special token whose data is the time at which the token was generated. The default internal structure of the trigger generator component is a single output port. Tokens are produced on the port according to some internal parameters specified for the component, which include wait time, w, start modulus, m, and period, T. Usually, either w or m is specified, and once that time arrives a token is produced, and another token is produced every T seconds.

As a matter of implementation, empirical results from our work shows that using a `pthreads` [14] enabled operating system[2] (but not a real-time OS) results in a variance in expected time generation of approximately 2-3 milliseconds. On a real-time OS[3], the variance is less than 1 ms.

3.3 Buffer Semantics

A buffer component, C_b provides an integer number of outputs, j, with inputs $k = j + 1$. The j output ports match to the j inputs of some existing component being buffered, C_a. Values, when received by an input, are queued by C_b. One particular input, the time trigger input, subscribes to the single output port of a trigger generator component. When

[2]Linux flavors Kubuntu and Gentoo were used.
[3]QNX was used for the RTOS.

a token is received on the time trigger port, the queued data values are sent to the output ports such that they can be received by C_a.

In future work, we may provide a special semantics for buffers where the most recent value received (only) on each input port is passed to the output port. Our current semantics requires that if more than one value (or no value at all) is received by the buffer between triggers, then C_a is responsible for determining whether to use all, or only the most recent, values.

4. TRANSFORMATION AND EXAMPLE

Given the advantages of a system whose timing characteristics are explicitly expressed, we describe now a transformation from a purely event-driven model to one with time-triggered execution. This transformation modifies an existing graph, and permits existing components to execute with no behavioral changes: the only change to the system is the topological rewrite, and the insertion of new buffered components along with their time-based triggers. Our methodology is as follows: (1) examine an existing component interconnection graph; (2) insert, before each component of the graph, a time-triggered buffer; (3) insert, after each component of the graph, a time-triggered buffer; and (4) insert, somewhere in the graph, a timed event-generator for each buffer, which sends events at the appropriate time The rewriting rules for this methodology are trivial, when specified using the GReAT rewriting language [2]. In this section, we provide a subset of the transformations required.

4.1 Domain-specific Modeling Language

Our systems are defined using a domain-specific language, capable of synthesizing experiments based on component interconnections. This work is explained thoroughly in [23]. The metamodel is shown in Figure 3.

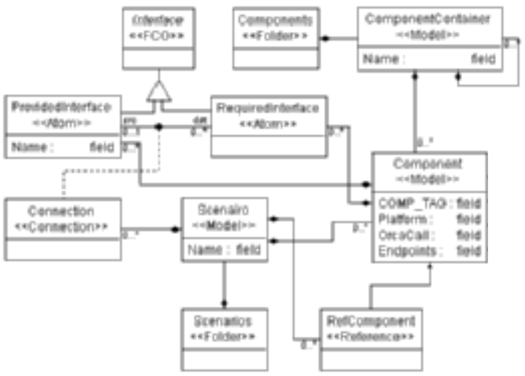


Figure 3: Relevant subset of the metamodel of our DSML (a screen capture from the GME tool).

Our system is built mainly of `Component` objects, connected to one another through provided and required `Interface` objects. The language provides several constraints (outside the scope of this paper) to prevent ill-formed models. We use this metamodel as the source *and* destination metamodel in the GReAT transformations.

4.2 Transformation Rules

The first task of the rewriting algorithm is to create new buffer components, whose job it is to implement the semantics described in Section 3. The precise semantics are generally decided for the entire graph, not piecemeal, and can be a value selected when the graph transformation is executed. For brevity, we have provided an abbreviated algorithm, that only inserts buffers for the *input* of each graph component. The algorithm to insert for component outputs is similar. An overall description of the rewrite is summarized by the rules shown in Figure 4.

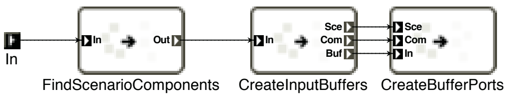


Figure 4: The order of rule execution for rewriting.

Buffers are added as `Component` objects, and the executable for this object can be generically synthesized based on the number of input/output ports, and the semantics chosen. We leave this detailed discussion to future papers, and concentrate instead on their insertion based on context. In Figure 5 the rule details are shown.

The rule can be read as follows: for each `Component` that contains a `RequiredInterface` object, create a new `InputBuffer:Component`, with a `RequiredInterface` to accept time triggers, that will become its input buffer. Create also a new `Trigger:Component` with a `ProvidedInterface` that will provide time triggered events, and connect that provided port to the required trigger in the new buffer. Various objects are renamed for clarity in the final model.

We must also replace existing connections between two `Component` objects by routing those connections between the new `InputBuffer:Compoment` that we created above. A similar rule (not shown for brevity) removes the existing `Connection` and creates two new ones, such that the `InputBuffer:Component` maps the data through to the `Component`.

Other rules not shown insert the necessary configuration items for each trigger component, as discussed in Section 3. These include the frequency of execution (based on the WCET of the component), and the time modulus at which to start running.

Executing all of the rules gives the transformation result shown in Figure 6.

5. ANALYSIS AND DISCUSSION

What role does Domain-Specific Modeling play in this application of transformations? In fact, why is modeling useful as a design concept here, rather than just using "old-fashioned" programming to solve the problem? What is the particular advantage that model transformations give to enable this migration from event-based to time-based behaviors?

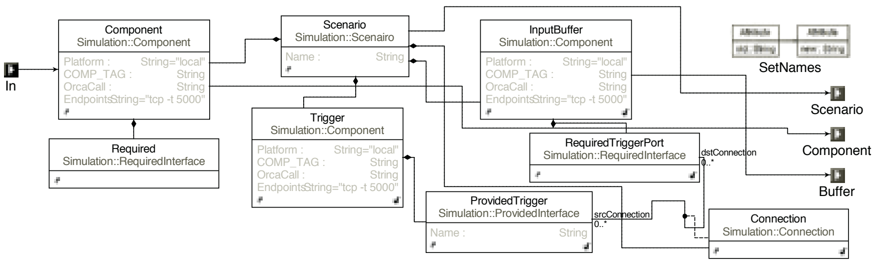
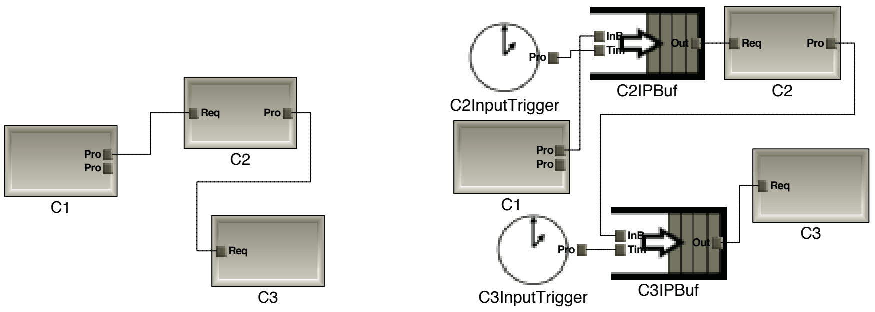


Figure 5: GReAT rule to create buffer components for triggered input reading based on time-triggered events. Note that the trigger scheduler is generated at the same step.



(a) Original model.

(b) Transformed model.

Figure 6: (a) An original model, with only event-based triggering. (b) The result of transforming (a) with the rules discussed in Section 4.2.

With response to the relation to DSM, the application requires a domain-specific encoding of existing solution. The language we designed places particular emphasis on event-based behaviors, and the simple interfaces defined between components of production and consumption enable the direction of data to be well understood. Thus, such a language is able to capture existing systems, *and run them* [23]. This is an important distinction: a design that concentrated mainly on how the *new* system, with its new semantics, should be represented would be unable to validate that an existing model accurately represents the system as is.

So, why not use just clever programming techniques to migrate systems to the new semantics? Certainly this approach could be taken, but at the usual risk of requiring experts in the system design to become experts in new semantics, and new techniques for execution. Our approach permits existing system components to *continue to work using their old semantics*. Thus, not one line of code needs to be changed in the existing software to conform to the new time-triggered behaviors.

Additionally, system experts know how their systems are currently organized, and implemented. These experts can take our language and represent the models both as they are, and (perhaps) as they *should* be in the future. The model transformation approach to creating the new buffers is advantageous in that it reuses the investment in the encoding of the system into our DSML.

The model transformation approach (through GReAT) reuses the metamodel-based specification of the DSM in a way that language designers can discuss how types are used, and new components are generated, with the system experts. We again point out that this permits reuse of existing code-bases. Approaches that do not use the strong typing and constraint-based organization that DSMLs provide to the end user run the risk that some corner cases may not be covered, or that assumptions made by the migration software are invalid according to the metamodel.

6. CONCLUSION AND FUTURE WORK

We have described how event-driven component-based systems can experience differences of execution based on subtle timing changes. We presented the notion of inserting time-triggered buffers as a way to reuse existing component code, while increasing the timing accuracy of component execution. We showed how, with an existing DSML to model such component-based systems, we can use model transformation techniques to enforce our time-triggered semantics on an existing model. We provided a semantics for these buffers, as well as the time triggers that control how long buffers hold their data tokens.

Our future work includes various autogeneration of the buffer code for various component-based middleware frameworks. We can also become more sophisticated in the synthesis of time trigger components, to attempt to consolidate the necessary scheduling into a single component, rather than a distributed set of components, thus increasing the scalability of the system without increasing the number of components in the system linearly.

Acknowledgments

This work is supported by the Air Force Research Laboratory, under award #FA8750-08-1-0024, titled "MultiCore Hardware Experiments in Software Producibility."

7. REFERENCES

[1] J. Auerbach, D. F. Bacon, D. T. Iercan, C. M. Kirsch, V. T. Rajan, H. Roeck, and R. Trummer. Java takes flight: time-portable real-time programming with exotasks. *SIGPLAN Not.*, 42(7):51–62, 2007.

[2] D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai. The graph rewriting and transformation language: GReAT. *Electronic Communications of the EASST*, 1, 2006.

[3] D. Box. *Essential COM*. Addison-Wesley, Reading, MA, 1997.

[4] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2-3):249–274, May 2000.

[5] E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent distribution of real-time components based on logical execution time. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 31–39, New York, NY, USA, 2005. ACM.

[6] G.Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress*. North-Holland Publishing Co., 1974.

[7] A. Goderis, C. Brooks, I. Altintas, E. A. Lee, and C. Goble. Heterogeneous composition of models of computation. Technical Report UCB/EECS-2007-139, EECS Department, University of California, Berkeley, Nov 2007.

[8] M. Henning and M. Spruiell. *Distributed Programming with Ice*. 3.3.1b edition, July 2009.

[9] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, Jan 2003.

[10] T. A. Henzinger, C. M. Kirsch, and S. Matic. Composable code generation for distributed Giotto. *SIGPLAN Not.*, 40(7):21–30, 2005.

[11] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. *Information Processing*, 1977.

[12] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, Oct. 2001.

[13] H. Kopetz and G. Grunsteidl. Ttp - a time-triggered protocol for fault-tolerant real-time systems. In *Proceedings of The Twenty-Third International Symposium on Fault-Tolderant Computing*, volume FTCS-23, 1993.

[14] B. Lewis and D. J. Berg. *Multithreaded Programming With PThreads*. Prentice Hall PTR, 1997.

[15] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 3rd edition, 2001.

[16] Object Modeling Group. *Data Distribution Service for Real-Time Systems, Version 1.2*, formal/07-01-01 edition, January 2007.

[17] J. Prosise. *Programming Microsoft .NET*. Microsoft Press, June 15 2002.

[18] P. Puschner and C. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

[19] I. Pyarali, C. O'Ryan, D. Schmidt, N. Wang, A. Gokhale, and V. Kachroo. Using principle patterns to optimize real-time ORBs. *Concurrency, IEEE*, 8(1):16–25, Jan-Mar 2000.

[20] I. Pyarali, D. Schmidt, and R. Cytron. Techniques for enhancing real-time CORBA quality of service. *Proceedings of the IEEE*, 91(7):1070–1085, July 2003.

[21] R. E. Schantz, J. P. Loyall, D. C. Schmidt, C. Rodrigues, Y. Kirishnamurthy, and I. Pyarali. Flexible and adaptive QoS control for distributed real-time and embedded middleware. In *Proceedings of Middleware 2003*, Rio de Janeiro, Brazil, June 16-20 2003. 4th IFIP/ACM/USENIX International Conference on Distrubted Systems Platforms.

[22] D. Schmidt, D. Levine, and S. Mungee. The design and performance of real-time object request brokers. *Computer Communications*, April 1998.

[23] A. Schuster and J. Sprinkle. Synthesizing executable simulations from structural models of component-based systems. In *3rd International Workshop on Multi-Paradigm Modeling*, October 2009.

[24] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the java system. *USENIX Computing Systems, MIT Press*, 9(4), Nov/Dec 1996.

[25] K. Wong and C. Wang. Push-Pull Messaging: a high-performance communication mechanism for commodity SMP clusters. In *Parallel Processing, 1999. Proceedings. 1999 International Conference on*, pages 12–19, 1999.

[26] B. Zeigler, H. Praehofer, and T. Kim. *Theory of modeling and simulation*. Academic Press, 2000.