8th OOPSLA Workshop on Domain-Specific Modeling (DSM'08)

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

October 19-20, 2008
Nashville, TN USA
http://www.dsmforum.org/events/DSM08/

University of Alabama at Birmingham
Department of Computer and Information Sciences
1300 University Blvd.
Birmingham AL 35294
http://www.cis.uab.edu

ISBN: 978-0-61523-024-5

UAB Printing Solutions, Birmingham, Alabama

2008

Welcome to the 8th OOPSLA Workshop on Domain-Specific Modeling – DSM'08

Preface

Domain-Specific Modeling (DSM) has recently received a surge of interest due to its ability to raise the abstraction level of system development beyond programming by specifying the solution directly using domain concepts. DSM enables end-users to be participants in describing problems in their domain and can also improve the productivity of software developers. In many cases, final products can be generated automatically from these high-level specifications. This automation is possible because both the language and generators are aligned to the requirements of a specific domain. The abstractions that are available in domain-specific models allow reasoning and design at the appropriate level of abstraction using icons and idioms familiar to domain experts. For example, a domain-specific model 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 and underlying frameworks.

Continued investigation is still needed in order to further advance the acceptance and viability of domain-specific modeling. This workshop, which is in its eighth incarnation at OOPSLA 2008, features research and position 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. The two-day program contains demonstrations of DSM tools, a panel on evolution issues, and also a keynote.

We have organized the 16 papers in these proceedings to emphasize general areas of interest into which the papers loosely fit. In addition to examples of DSM, authors from both industry and academia present research ideas that initiate and forward the technical underpinnings of domain-specific modeling. As a whole the body of work highlights the importance of metamodeling, which significantly eases the implementation of domain-specific languages and provides support for experimenting with the modeling language as it is built.

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.

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

October 2008
Nashville, Tennessee USA

8th WORKSHOP ON DOMAIN-SPECIFIC MODELING

October 19-20, 2008 Nashville, Tennessee USA

Program Committee

Pierre America, Philips
Peter Bell, SystemsForge
Jorn Bettin, Sofismo
Philip T. Cox, Dalhousie University
Krzysztof Czarnecki, University of Waterloo
Brandon Eames, Utah State University
Ethan Jackson, Microsoft
Frederic Jouault, University of Nantes
Jürgen Jung, Deutche Post AG
Steven Kelly, MetaCase
Benoit Langlois, Thales
Gunther Lenz, Microsoft
Shih-Hsi Liu, California State University, Fresno
Kalle Lyytinen, Case Western Reserve University
Pentti Marttiin, Nokia Siemens Networks
Birger Moller-Pedersen, University of Oslo
Juha Pärssinen, VTT
Arturo Sanchez, University of North Florida
Markus Völter, independent consultant
Jing Zhang, Motorola Research

Organizing Committee

Juha-Pekka Tolvanen, MetaCase
Jeff Gray, University of Alabama at Birmingham
Matti Rossi, Helsinki School of Economics
Jonathan Sprinkle, University of Arizona

Keynote Summary

The Model Repository: More than just XML under version control
Domain-Specific Modeling: 20 years of progress?

Steven Kelly, Ph.D. - MetaCase

ABSTRACT: There is increased awareness within the modeling arena of the need for a central repository of system description information. This is brought on by a growing recognition that only with a strong central repository can modeling tools be integrated, cope with large projects, provide full life-cycle support, produce complete documentation, perform system-wide validation and verification, and adequately control a project. In examining the various approaches chosen or proposed by various tool providers and users vendors, it is apparent that for many a model repository is nothing more than an off-the-shelf version control system into which XML files are saved. However, as this talk will demonstrate, current version control systems and XML cannot be successfully employed as a model repository.

BIOGRAPHY: Dr. Steven Kelly is the CTO of MetaCase and co-founder of the DSM Forum. He has over ten years of experience of building metaCASE environments and acting as a consultant on their use in Domain-Specific Modeling. As architect and lead developer of MetaEdit+, MetaCase's domain-specific modeling tool, he has seen it win or be a finalist in awards from Byte, the Innosuomi prize for innovation awarded by the Finnish President, Net.Object Days, and the Software Development Jolt awards. Ever present on the program committee of the OOPSLA workshops on Domain-Specific Modeling, he co-organized the first workshop in 2001. He is author of over 20 articles, most recently in journals such as Dr. Dobb's and ObjektSpektrum, and is a member of the editorial board for the Journal of Database Management. He has an M.A. (Hons.) in Mathematics and Computer Science from the University of Cambridge, and a Ph.D. from the University of Jyväskylä. His computer education began with machine code, Assembler and BASIC, and came to rest in Smalltalk. Outside of work, he has co-authored the first grammar of the Kenyan Orma language, and is a soccer player in the Finnish Third Division.

Table of Contents

Welcome message from the organizers i
List of program and organizing committees ii
Keynote Summary iii
Table of Contents iv

A Comparison of Tool Support for Textual Domain-Specific Languages 1
Michael Pfeiffer and Josef Pichler
Undoing Operational Steps of Domain-Specific Modeling Language 8
Tim Hartmann and Daniel A. Sadilek
The Interchange of (Meta)Model between MetaEdit+ and Eclipse EMF using M3-Level-based Bridges 14
Heiko Kern
Foundations for a Domain Specific Modeling Language Prototyping Environment: A Compositional Approach 20
Luis Pedro, Didier Buchs and Vasco Amaral
A Domain-Specific Approach to the Development of Ontology-Based Document Assessment Assistants 28
Arturo J. Sánchez-Ruíz and Bart Welling
A Common Meta-Model for Data Analysis based on DSM 35
Yvette Teiken and Stefan Flöring
Towards Model-Based Testing of Domain-Specific Modelling Languages 39
Janne Merilinna, Olli-Pekka Puolitaival and Juha Pärssinen
ModelTalk: A Framework for Developing Domain Specific Executable Models 45
Atzmon Hen-Tov, David Lorenz, and Lior Schachter
SMML: Software Measurement Modeling Language 52
Beatriz Mora, Felix Garcia, Francisco Ruiz and Mario Piattini
The Transformation-Driven Architecture 60
Janis Barzdins, Sergejs Kozlovics and Edgars Rencis
When Frameworks Let You Down: Platform-Imposed Constraints on the Design and Evolution of Domain-Specific Languages 64
Danny Groenewegen, Zef Hemel, Lennart C. L. Kats and Eelco Visser
Visual Specification of DSL Processor Debuggers 67
Tamás Mészáros and Tihamer Levendovszky
A Domain Specific Design Tool for Spacecraft System Behavior 73
Sravanthi Venigalla, Brandon Eames and Allan McInnes
Using Integrative Modeling for Advanced Heterogeneous System Simulation 80
Tapasya Patki, Hussain Al-Helal, Jacob Gulotta, Jason Hansen and Jonathan Sprinkle
A Family of Languages for Architecture Description 86
Markus Völter
Domain-Specific Modelling Language for Navigation Applications on S60 Mobile Phones 94
Janne Merilinna

A Comparison of Tool Support for Textual Domain-Specific Languages

Michael Pfeiffer
Software Competence Center Hagenberg
michael.pfeiffer@scch.at

Josef Pichler
Software Competence Center Hagenberg
josef.pichler@scch.at

Abstract

A domain-specific language is a specialized and problem-oriented language. Successful application of a DSL largely depends on provided tools, so called language workbenches that support end-programmers creating, editing, and maintaining programs written in a DSL. In this paper, we describe four different tools supporting the creating of language workbenches, identify commonalities and differences and compare these tools by means of a set of criteria.

1. Introdcution

Domain-specific languages (DSL) are languages tailored to a specific application domain [1]. They have the potential to reduce complexity of software development by raising the abstraction level towards an application domain. According to the application domain, different notations (textual, graphical, tabular) are used. Textual languages have long tradition in the domain of textual meta-modeling languages like PSL/PSA [2] used for analysis and documentation of requirements and functional specifications for information systems. In this paper, we focus on textual languages following the source-to-source transformation (see [3]), i.e. DSL code is compiled into a general purpose programming language like Java and C#. Such textual DSLs are applied successfully in various areas. For instance, Monaco [4] is a language for end-user programming for automation systems and Tbl [5] is a language for test scripts of mobile devices. Successful application of such languages largely depends on provided tool support. A tool that support end-programmers creating, editing, and maintaining programs in a DSL is called a language workbench [6]. Users expect workbenches including full-featured editors with syntax highlighting, code completion, etc. as available for general programming languages. Such editors are often implemented manually resulting in effort and costs that can be reduced only to a certain degree by using editor frameworks, for example the Eclipse platform. Furthermore, language workbenches require further components like lexical scanner, parser, and data structures for the abstract syntax tree as well as generators to transform the program to a general programming language.

Both industry and research community have recognized this circumstance so that today several tools exists that allows automatically generation of such language workbenches based on meta-models or grammar definitions. The automatically generation of a compiler frontend including scanner and parser out of a context-free grammar is an established area and a lot of such tools like CoCo/R [7] or Antlr [8] are available today. The additional generation of editors towards a full-fledged language workbench is a consequent further development.

In this paper, we compare four different tools supporting the generation of language workbenches for textual DSLs. In particular, we describe and compare approaches for definition of the language, used techniques for parsing and code generation as well as the functional range of the resulting workbenches. For illustration of commonalities and differences, we created language workbenches for a simple language used to define finite state machines. Furthermore, we provide comparison based on a subset of criteria defined by the feature model of DSL tools given in [9].

This paper is organized as follows. Section 2 defines criteria for comparison. Section 3 holds the language definition for our example language for finite state machines. Section 4 gives an overview of tools supporting textual DSL. Section 5 compares these tools based on the defined criteria.

2. Criteria for Comparison DSL Support

The goal of this section is to describe the criteria to be used for the comparison of selected DSL tools. The criteria are based on the DSL feature model defined by Langlois et al. [9]. This feature model covers languages, transformation, tooling, and process aspects. Instead of defining a new criteria catalog, the reuse of an existing catalog facilitates the comparison of our results with other comparisons too, in particular, with the comparison of graphical DSL tools given in [9].

As our comparison will mainly focus on tool support of textual DSL, we omitted some criteria. For example, the process aspects that are defined as optional features only are omitted. Hence, the criteria to be used may be split into three groups:

- Language (LA)
- Transformation (TR)
- Tool (TO)

The criteria for each group and the difference to the feature model are explained at a glance below. The abbreviation of the group combined with a criteria identifier is unique and used later to compactly describe the results. In contrast to the feature model in [9], the criteria will be explained by means of questions; this is a very straightforward and simple approach and should clearly show what the respective criteria are about.

2.1 Language

This group of criteria comprises tool support for both the abstract syntax (AS) and concrete syntax (CS).

- LA-AS1. Which representation is used for the abstract syntax (abstract syntax tree or abstract syntax graph)?
- LA-AS2. Which representation is used for the definition of the abstract syntax (grammar or a meta-model)?

- LA-AS3. Can the abstract syntax be composed of several grammars or meta-models?
- LA-CS1. Which technique is used to map abstract syntax to concrete syntax?
- LA-CS2. Which representation (text, graphic, wizard, or table) can be used for the concrete syntax?
- LA-CS3. Which style (declarative or imperative) can be used for the concrete syntax?

As our focus is on textual tools, the criteria LA-CS2 will be evaluated as text for all tools.

2.2 Transformation

The criteria of this group have to answer questions about specification of transformation, expected target assets, and the realization to produce the expected target assets. The transformation realizes the correspondence from the problem to the solution space. A target asset is a software artifact resulting from the transformation. The criteria of this group cover target assets (TA) and the operational transformation (OT).

- TR-TA1. Which representations of the target asset (model, text, graphic, binary) are possible?
- TR-TA2. Which kind of support of asset update (destructive or incremental) is possible?
- TR-TA3. Which kind of support for integration of target assets is used?
- TR-OT1. Which kinds of transformation technique are used (model-to-model (M→M), model-to-text (M→T), T→T, T→M)?
- TR-OT2. Which mode (compilation or interpretation) is used for transformation execution?
- TR-OT3. Which environment (internal or external) is used for transformation execution?
- TR-OT4. Which scheduling (implicit or explicit) form is used?
- TR-OT5. Which location (internal or external) is used?
- TR-OT6. Which automation level (manual or automated) is used?

The criteria variability and phasing of the feature model are omitted.

2.3 Tool

This group of criteria stresses on overall tool support.

- TO-RA1. Which respect of abstraction (intrusive or seamless) is used?
- TO-AS1. Which kind of assistance (static or adaptive) is provided?
- TO-AS2. Which kind of process guidance (step or workflow) is provided?
- TO-AS3. Which kind of checking (completeness or consistency) is supported?

Criteria of the feature model covering quality factors like reliability and efficiency are omitted because they are not specific to DSL tools and, hence, do not make a substantial contribution for our comparison.

3. Example

For illustration of commonalities and differences of tools in the next section, we show examples based on a simple language used to define finite state machines (FSM). For our example, a FSM will be described in a text file following a simple syntax. The target asset is Java source code. The lexical and syntactical structure of FSM text files are defined by the following grammar given in the EBNF proposed by Wirth [10]:

```
FSM        = "inputAlphabet" string
               "outputAlphabet" string {State}.
State      = ["start"] "state" id {Transition}.
Transition = "transition" char ["/" char] "->" id.
```

A FSM is described by the input alphabet, the output alphabet, and a set of states. A state has an identifier and a set of transitions that connect a state to following states. A transition is composed of an input character that triggers the transition, an optional output character, and the identifier of the following state.

Terminal classes *id*, *string*, and *char* of the FSM are specified as regular expressions:

id = [:alpha:][:alnum:]+ *string* = "[~ \t\n\r]*" *char* = [~ \t\n\r]

The following example shows a finite state machine that determines if a binary number has an odd or even number of zero digits.

```
// Digits of a binary number
inputAlphabet "01"
// Char 'e' for even and 'o' for odd
outputAlphabet "eo"
start state Even
transition 0 / o -> Odd
transition 1 / e -> Even
state Odd
transition 0 / e -> Even
transition 1 / o -> Odd
```

Even this example is very simple it suffices to demonstrate most important aspects of different tools and differences between these tools.

4. Overview of Selected Tools

In this section, we give an overview of tools supporting textual DSLs. According to the focus of this paper, a first investigation resulted in following tools to be considered:

- openArchitectureWare (oAW), version 4.3
- Meta Programming System (MPS), early access version
- MontiCore, version 1.1.5
- IDE Meta-Tooling Platform (IMP), version 0.1.74
- Textual Concrete Syntax (TCS), version 0.0.1
- Textual Editing Framework (TEF), version 1.0.3
- CodeWorker, version 3.5

For the selection of tools to be included, we considered to omit tools that are very similar (at least on technology view) to other ones or system that cannot be considered to be used in real projects (for example, due to immature academic tool). For example, oAW, TCS, and TEF are based on same technology (Eclipse, GMT); hence, the systems TCS and TEF were omitted in favor of the mature oAW. The CodeWorker tool was not considered because it did not provide editor support at the time of the evaluation.

```
fsm.xtxt
// Finite State Machine

FSM :
    "inputAlphabet" inputAlpahbet = STRING
    "outputAlphabet" outputAlphabet = STRING
    (states += State)*
;

State :
    (isStart ?= "start")?
    "state" (name = ID)
    (transitions += Transition)*
;

Transition :
    "transition" input = CHARACTER
    ("/" output = CHARACTER)?
    "->" (state = [State])
;

Native CHARACTER : "~('\n'|'\r'|' '|'\t'|'\\u000C')";
```

Figure 1. xText grammar editor.

4.1 openArchitectureWare

The openArchitectureWare [11] [12] is an open source project for model-driven software development, based on the Eclipse platform. The sub-component xText provides a framework for textual languages, whereas the resulting DSL environment is also based on the Eclipse platform.

The concrete syntax is specified as context-free grammar including production rules and types of terminal symbols. Figure 1 shows the grammar of our FSM example in the corresponding editor provided by oAW. The schema of the grammar supports cross references, type inheritance, and enumerations. The corresponding editor supports syntax highlighting, code completion of keywords and meta-model elements, validation and multi-file templates among other things.

The grammar contains sufficient data to generate the main building blocks of a language workbench supporting the specified language including:

- an abstract syntax graph (ASG)
- scanner and parser for text-to-model transformation by means of Antlr [8]
- a generator for model-to-text transformation by means of xPand part of oAW
- an editor based on the Eclipse

The generation of all these artifacts is configured and controlled by a so-called workflow definition file, as shown in Figure 2.

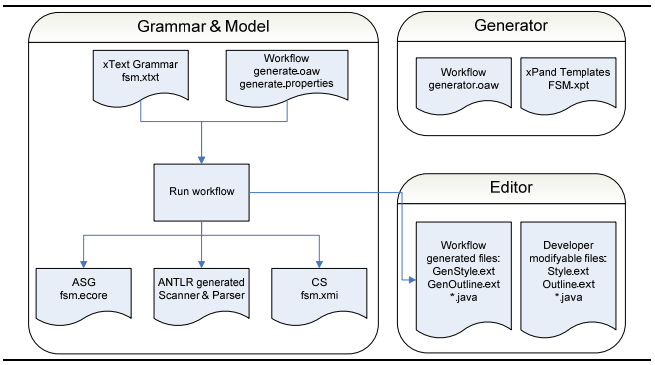


Figure 2. Overview oAW.

```
*model.fsm
// Output alphabet is 'e' for even and 'o' for odd
outputAlphabet "eo"

start state Even
transition 0 / o -> Odd
transition 1 / e -> Even
tran
    start
    transition
stat
tran
Problems
2 errors, 1 w
Description
Erro
War
```

Figure 3. oAW FSM editor.

The generated artifacts are separated into files (Java source code or configuration files) that are indented to be reworked and modified by the developer and files that are regenerated each time the workflow is run. This separation avoids a common problem [13] of synchronization of single files that are both generated automatically but also reworked by the developer.

The generated ASG in form of an Ecore model is based on the Eclipse Modeling Framework [14]. oAW uses Antlr to generate scanner and parser for text-to-model transformation whereas for model/text transformation, the template engine xPand, part of oAW, is used.

The generated text editor for manipulation of DSL texts supports syntax highlighting, code completion, validation of syntax and model constraints checking. Figure 3 shows an example FSM opened in the editor whereas a syntax error is showed together with provided code completion in order to fix the error.

At the current state, oAW has some limitations of the expressiveness of a DSL grammar. However it provides good tools with early validations to quickly develop a DSL with semantic checks and text generator.

4.2 Meta Programming System

The Meta Programming System (MPS) [15] [16] of JetBrains has not been released yet; however, an evaluation version can be obtained in an early access program. Nonetheless, we have included the MPS to our comparison because of its unique technique to define the syntax of a DSL and the used cell-based editing model.

The abstract syntax tree (AST) is specified by a list of so-called concepts. In MPS, all concepts of one language together are called the structure of the language.

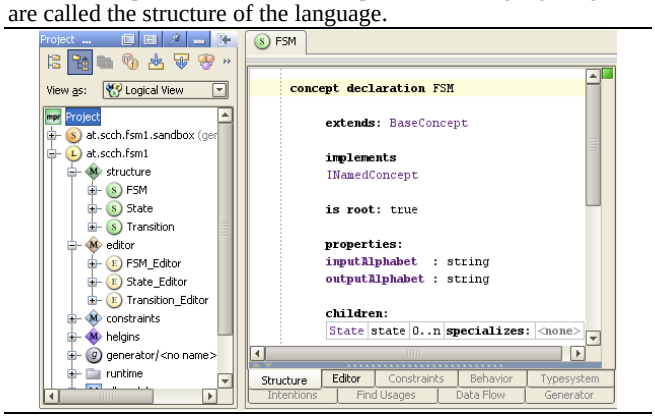


Figure 4. MPS concept editor.

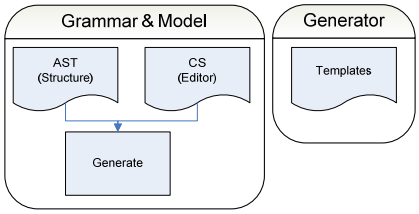


Figure 5. Overview MPS.

Figure 4 shows the concept for the top-level element FSM of our example described in section 3. A concept can extend another concept to inherit its properties. For instance, the concept *FSM* defined Figure 4 extends the built-in concept *BaseConcept*.

Concepts are connected together by means of references or aggregations. An example for an aggregation is the connection between the *FSM* concept and the *State* concepts. Furthermore, a concept has properties that are typically used to contain the values of terminal symbols. One concept of a language must be declared as root element.

The concrete syntax of a concept has to be defined in form of static text that is not editable and editable cells. An example is given later on in this section.

Figure 5 gives an overview of the affected models for the implementation of the example described in section 3. MPS provides a generator following a model-to-model approach that allows the transformation from any model to any other model. The transformation is described by means of templates that are edited with a cell editor.

Figure 6 shows the resulting cell editor for our FSM language containing the example FSM defined in section 3. The editing area is separated into read-only areas and editable cells. For instance, the properties of the third transition in the example are editable, whereas the text outside is read-only. Manipulation of texts follows always the same pattern: first, the kind of element to be added or edited must be selected and, second, the properties for that element can be edited in so called cell editors.

MPS is interesting as it provides a self-contained complete solution that uses its own technology every where, for example editors for manipulation of syntax or templates follow the same approach as the resulting editor for our DSL. For another MPS example we refer to an article by Martin Fowler [16].

4.3 MontiCore

The MontiCore framework [17] [18] [19] is a research project by Software Systems Engineering Institute, TU Braunschweig, Germany. Its core and the generated editor for the DSL are based on the Java/Eclipse platform.

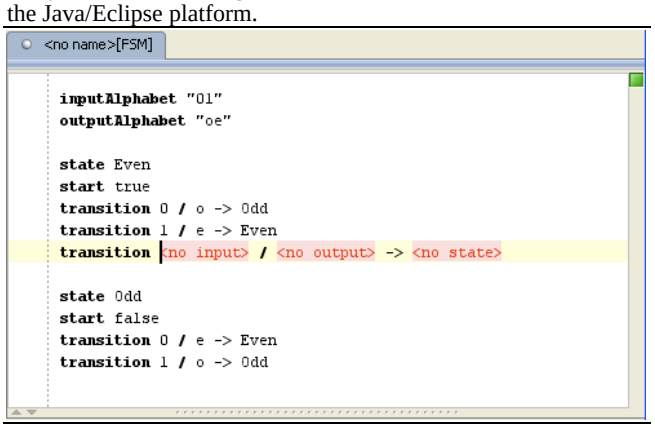


Figure 6. MPS FSM editor.

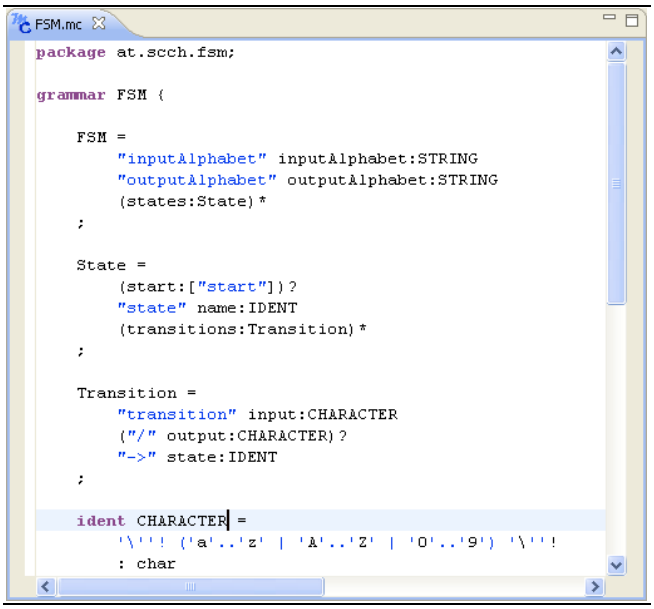


Figure 7. MontiCore grammar editor.

MontiCore uses an enriched context-free grammar for the specification of the concrete syntax which is similar to the input format of Antlr [8]. In addition to rules of the grammar, it contains definitions of the data types of terminal symbols, whereas the whole range of simple Java data types is supported.

Figure 7 shows the MontiCore editor with the grammar described in section 3. According to the used input format, the grammar differs slightly from the grammar described in section 3. The input and output character are now enclosed within two apostrophes. Furthermore, MontiCore does not accept a terminal symbol that consists of a single character only. MontiCore uses a single source for defining concrete and abstract syntax of a DSL [18] [20]. The grammar contains sufficient data to generate data structures for the abstract syntax tree as well as scanner and parser, as shown in Figure 8. The later ones are generated by means of the compiler-generator Antlr. However, the parser must be connected with the editor for validation manually by implementing some glue code in the Java programming language.

The model transformation can either be realized in form of Java code using the visitor pattern or using the provided template engine. The model transformation can be triggered by the user inside the editor.

Figure 9 shows the resulting editor for our FSM language with an example FSM. As shown by the example, the syntax is almost as specified in section 3. Working with MontiCore requires labor however it is also more flexible in the expressiveness of the DSL.

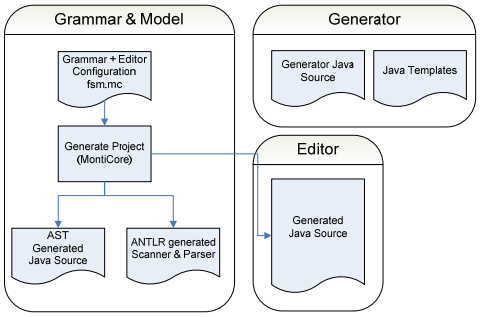


Figure 8. Overview MontiCore.

```
*example.fsm
7  inputAlphabet "01"
8  outputAlphabet "oe"
9
10 start state Even
11 transition '0' / 'o' -> Odd
12 transition '1' / 'e' -> Even
13 tran
14
15 state Odd
16 transition '0' / 'e' -> Even
17 transition '1' / 'o' -> Odd
```

Figure 9. MontiCore FSM editor.

4.4 IDE Meta-Tooling Platform

The IDE Meta-Tooling Platform (IMP) [21] is an open source project begun at IBM Watson Research. Its goal is to ease the development of IDE support for a new programming language. IMP provides wizards to generated code skeletons for a large range of features required for an IDE of a new language.

The concrete syntax of a language is defined by a context-free grammar, whereas the grammar is divided into input used to generate the scanner (lexer) and the parser. Figure 10 shows the input for the parser for our example language edited in a plain text editor. Productions of the grammar can be annotated with Java code that is added to the data structures for the abstract syntax tree.

IMP includes the parser generator LPG [22] to generate the data types for the abstract syntax tree as well as scanner (lexer) and parser, as shown in Figure 11. Furthermore, IMP generates a full-featured editor based on the Eclipse platform including outline and syntax highlighting, whereas the editor includes the generated parser. The resulting editor is registered in the Eclipse platform using extension points. IMP utilizes the visitor pattern for model transformations. For example a source code formatter can be implemented using the visitor pattern.

Figure 12 shows the editor with the example FSM. IMP provides a generic text editor that supports syntax highlighting, folding, formatting and code completion. An outline of the text in the editor is provided too. IMP does not include a template engine so other solutions have to be used instead. For our example, the code generator was implemented by means of JET [23].

```
FSMParser.g
%Rules

    FSM ::=
        inputalphabet Text
        outputalphabet Text
        States

    States ::= %Empty
        | State States

    State ::= StartOp state Identifier Transitions

    StartOp ::= %Empty
        | start

    Transitions ::= %Empty
        | Transition Transitions

    Transition ::= transition Input OutputOp
        "->" TransitionState

    Input ::= Char
```

Figure 10. IMP parser grammar editor.

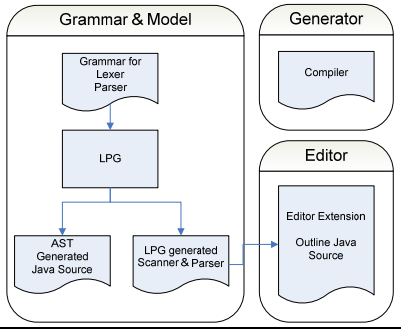


Figure 11. Overview IMP.

As the goal of IMP indicates its usage is not limited to the construction of DSLs. Compared to the other tools it requires more programming effort but on the other hand provides also more flexibility.

5. Comparison

In this section, we will compare the tools, described in the previous section, by means of the criteria defined in section 2. Figure 13 contains the result of the comparison at a glance that is described in more detail below.

5.1 Language

For the representation of the abstract syntax, both trees (AST) and graphs (ASG) are used and realized with different strategies. *MontiCore* and *IMP* generate Java source code implementing the abstract syntax tree, whereas *oAW* and *MPS* use generic meta-models for the abstract syntax graph. Except for *IMP*, the definition of the abstract syntax can be composed of several grammars or meta-models.

All tools differ in the mapping between concrete syntax and abstract syntax. The tools *oAW* and *MontiCore* use a single source for defining both the concrete and the abstract syntax. In *MPS*, a user first defines the abstract syntax in form of concepts and, afterwards, he defines the concrete syntax for every concept. By contrast, *IMP* requires the definition of the concrete syntax only, and the abstract syntax is derived automatically from the concrete syntax.

All tools have in common, that the concrete syntax is represented as text. Of course, this is a consequence of the focus of this paper on tools for textual languages. However, MPS stores the model as XML document and presents it as text in the editor only. All tools support both declarative and imperative style of languages.

```
*example.fsm
// Output alphabet is 'e' for even and 'o' for odd
outputAlphabet "eo"

start state Even
transition 0 / o -> Odd
transition 1 / e -> Even
tran

state Odd
transition 0 / e -> Even
```

Figure 12. IMP FSM editor.

5.2 Transformation

All tools allow the generation of text files as target assets, however, the support level differs. *oAW* provides an outstanding transformation support. The template editor of *oAW* provides code completion and early error detection. *MPS* support model-to-model transformation out-of-the-box, so text generation requires generation of a target model that, in turn, can be transformed into text. *MontiCore* provides a rudimentary template engine only. The invocation of the template engine requires to write Java code for each text file to be generated, though. *IMP* is the only tool that has no transformation support built-in, so transformation must be implemented in the Java programming language following the visitor pattern. All tools provide destructive update of generated assets only.

All tools except *IMP* provide model-to-text mappings. Additionally, *MontiCore* and *MPS* provides model-to-model mappings too. For *MPS*, the model-to-text mapping requires an intermediate model-to-model mapping.

Concerning operational translation covered by criteria TR-OT2 – TR-OT6, all tools are very similar. Three tools follow an interpreted approach whereas a template engine fills templates at runtime; only *IMP* requires compilation of visitor classes implemented in the Java programming language. *MPS* allows starting the transformation process in the same environment whereas all other tools require a new instance of the Eclipse workbench. Only IMP schedules the transformation automatically after changing the DSL text; other tools require manual scheduling of the transformation. Finally, all tools use internal execution of the transformation.

5.3 Tool

Regarding tool assistance we observed a high variance ranging from a plain text editor for CS definition (*IMP*) to an editor with syntax coloring (*MontiCore*), code completion and validation while typing (*oAW* and *MPS*).

A well supported template editor is also very important. Support for editing templates range from using an existing editor without special template support (*MontiCore*), to an editor with comprehensive template support (*oAW* and *MPS*). The latter provides syntax highlighting, code completion and validation.

6. Conclusion

In this paper, we have described four different tools supporting textual DSLs. Furthermore, we compared these tools by a set of criteria based on the feature model of Langlois et al. [9]. The reuse of the criteria facilitates comparison of results presented in this paper with other comparisons, for example the comparison of graphical DSL tools given in [9] and [24].

The feature-set of the resulting language workbenches, mainly the editor, ranges from a plain text editor to a full-featured editor with syntax coloring, code completion and validation while typing.

All tools except *MPS* generate language workbenches based on the Eclipse platform. This explains the commonalities of the resulting workbenches. *MPS* is unique in this comparison as its editor is cell based instead of free text used by the other tools.

References

[1] M. Mernik, J. Heering, A. Sloane. When and How to Develop Domain-Specific Languages, *ACM Computing Surveys*, vol. 37, no. 4, pp. 316-344, December 2005.

[2] D. Teichroew and E.A. Hersehey. PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems. *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, 41-48, 1977.

[3] D. Spinellis. Notable design patterns for domain-specific languages. *The Journal of Systems and Software* 56, 91-99, 2001.

[4] H. Prähofer, D. Hurnaus, C. Wirth, H. Mössenböck. The Domain-Specific Language Monaco and its Visual Interactive Programming Environment. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Coeur d'Aléne, Idaho, USA, September 23-27, 2007.

[5] W. Hargassner, T. Hofer, C. Klammer, J. Pichler, G. Reisinger. A script-based testbed for mobile software frameworks. In *Proceedings First International Conference on Software Testing, Verification, and Validation*, pp. 448-457, 2008.

[6] M. Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages*?. 2005. http://martinfowler.com/articles/languageWorkbench.html.

[7] H. Mössenböck. *Coco/R for various languages – Online Documentation*. http://www.ssw.uni-linz.ac.at/Research/Projects/Coco.

[8] T. Parr and R. Quong. ANTLR: A Predicated-LL(k) Parser Generator. In *Journal of Software Practice and Experience*, 25(7):789-810, July, 1994.

[9] B. Langlois, C.E. Jitia, E. Jouenne. DSL Classification. In 7th *OOPSLA Workshop on Domain-Specific Modeling*, 2007.

[10] N. Wirth. What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? *Communications of the ACM*, November 1977.

[11] *Open Architecture Ware*, http://www.openarchitectureware.org.

[12] P. Friese, S. Efftinge, J. Köhnlein. *Build your own textual DSL with Tools from the Eclipse Modeling Project*. 2008. http://www.eclipse.org/articles/article.php?file=Article-BuildYourOwnDSL/index.html.

[13] L. Angyal, L. Lengyel, H. Charaf. A Synchronizing Technique for Syntactic Model-Code Round-Trip Engineering. *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the* Engineering of Computer Based Systems, pp.463-472, April, 2008.

[14] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. Grose. *Eclipse Modeling Framework*. Addison-Wesley, August 2003.

[15] *Meta Programming System*. http://www.jetbrains.com/mps.

[16] M. Fowler. *A Language Workbench in Action – MPS*. 2005. http://martinfowler.com/articles/mpsAgree.html.

[17] *MontiCore* http://www.monticore.de.

[18] H. Grönninger, H.Krahn, B. Rumpe, M. Schindler, S. Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *Proceedings of the 30th International Conference of Software Engineering (ICSE)*, Leipzig, Germany, 2008.

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

[20] H. Krahn, B. Rumpe, S. Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Proceedings of the ACM/IEEE 10th International Conference of Model Driven Engineering Languages and Systems* (*MODELS*), Nashville, TN, USA, 2007.

[21] *IDE Meta-Tooling Platform*. http://www.eclipse.org/imp.

[22] *The LALR parser generator* (*LPG*). http://sourceforge.net/projects/lpg.

[23] JET, http://www.eclipse.org/modeling/m2t/?project=jet.

[24] T. Özgür. Comparison of Microsoft DSL Tools and Eclipse Modeling Frameworks for Domain-Specifc Modeling. *Master Thesis, Software Engineering, MSE-2007:07, Bleking Institute of Technology*, Sweden, 2007.

	oAW	MontiCore	MPS	IMP
Language				
LA-AS1	ASG	AST	ASG	AST
LA-AS2	ECore meta-model	Java Classes	Proprietary meta-model.	Java Classes
LA-AS3	Composition.	Composition.	Composition.	No built-in support.
LA-CS1	Explicit. CS definition mixed with AS definition.	Explicit. CS definition mixed with AS definition.	Explicit. For each element of AS editor layout of CS is defined.	Implicit. CS defines AS.
LA-CS2	Text.	Text.	Text. Model instance stored in XML file.	Text.
LA-CS3	Declarative or imperative depends on decision of DSL developer.			
Transformation				
TR-TA1	Text. Template engine creates text files.	Text. Template engine creates text files.	Model or Text. Generator creates model instance. That in turn can generate text.	No built-in support.
TR-TA2	Destructive. Overwrites TA.	Destructive. Overwrites TA.	Destructive. Overwrites TA.	No built-in support.
TR-TA3	No integration support available.			
TR-OT1	M2T using template engine (xPand).	M2M. Using visitor pattern. M2T. Using visitor pattern or template engine.	M2M. Model to model generator. M2T. Using a target language that generates text (e.g. BaseLanguage generates Java Code).	No built-in support.
TR-OT2	Interpretation. Templates filled at runtime.	Interpretation. Templates filled at runtime.	Interpretation. Templates filled at runtime.	Compilation if using visitor pattern.
TR-OT3	External. Runtime workbench must be launched.	External. Runtime workbench must be launched.	Internal. Editor and Transformation in MPS.	External. Runtime workbench must be launched.
TR-OT4	Explicit. Workflow triggered by user.	Explicit. Triggered by user.	Explicit. Triggered by user.	Implicit. Eclipse builder runs after change.
TR-OT5	Internal. Runs in runtime workbench.	Internal. Runs in runtime workbench.	Internal. Runs in MPS.	Internal. Runs in runtime workbench.
TR-OT6	Manual. Triggered by user.	Manual. Triggered by user.	Manual. Triggered by user.	Automatically after change.
Tool				
TO-RA1	Depends on DSL. Our example DSL is seamless.			
TO-AS1	Adaptive. Code completion. Validation.	Adaptive. Validation.	Adaptive. Code completion. Validation.	Adaptive. Validation.
TO-AS2	Neither step nor workflow process guidance is supported.			
TO-AS3	Completeness. Using constraint language. Consistency. Ensured by grammar validation.	Completeness. Needs implementation. Consistency. Ensured by grammar validation.	Completeness. Using constraint language. Consistency. Grammar validation.	Completeness. Needs implementation. Consistency. Ensured by grammar validation.

Figure 13. DSL tool criteria comparison.

Undoing Operational Steps of Domain-Specific Modeling Languages

Tim Hartmann Daniel A. Sadilek

Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin, Germany
{hartmann|sadilek}@informatik.hu-berlin.de

Abstract

In this paper, we deal with the animated execution of domain-specific models (DSMs) that are expressed in domain-specific modeling languages (DSMLs) whose semantics are described in an operational fashion. We propose to support stepping back in the execution history of such DSMs. We argue that this eases debugging of the DSM itself and the DSML's operational semantics. As an example, we show animated model execution of Petri nets and identify the requirement to step back in their execution history. To accomplish this, we present an approach in which we apply principles for undoing user input in model editors to the animated execution of DSMs. Finally, we present an Eclipse-based implementation of our approach, which is an extension of the tool EPROVIDE.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

General Terms Design, Languages

Keywords Domain-specific modeling languages, operational semantics, debugging, stepping back, Eclipse

1. Introduction

1.1 Domain-Specific Modeling Languages

In contrast to general-purpose languages like the Unified Modeling Language (UML), domain-specific modeling languages (DSMLs) are customized to a particular application domain. With DSMLs, domain experts can create models using a vocabulary they are used to. These models can be directly used in a software engineering process. For example, they can be interpreted or code can be generated from them.

When developing DSMLs, the requirements are not always clear from the start, and more than one development iteration is necessary. In such cases, prototyping of DSMLs is needed. In this paper, we deal with executable DSMLs. A prototyping process for such DSMLs requires that domain experts can create and execute example models expressed in the DSML.

1.2 Domain-Specific Model Execution using Operational Semantics

When a domain-specific model (DSM) is to be executed, the DSML it is expressed in must be given execution semantics. This can be done with a *translational* or an *interpretational* approach.

Using the translational approach, model execution is prepared by *translating* a model into an executable form (e.g., by code generation). But this shifts down the level of abstraction (e.g., into the realm of programming languages). It is, therefore, inappropriate for prototyping, especially when domain experts should contribute to the prototyping of the DSML.

Using the interpretational approach, models are executed by an interpreter. This interpreter can be hand-crafted or it can be based on an executable description of the DSML's operational semantics. The operational semantics of a language describes the meaning of a language instance as a sequence of execution steps (Plotkin 1981). Generally, a transition system $\langle \Gamma, \rightarrow \rangle$ forms the mathematical foundation, where Γ is a set of *configurations* and $\rightarrow \subseteq \Gamma \times \Gamma$ is a *transition relation*.

Model-Driven Approach to Operational Semantics. Wachsmuth (2008) applies the idea of structural operational semantics to model-driven language engineering. The configurations in Γ are represented as models, which are called *configuration models*. Hence, the space of all possible configurations is defined with a metamodel, which is called *configuration metamodel*; and the transition relation $\rightarrow$ is defined with a model-to-model transformation, which is called *transition transformation*. As the transition transformation is an executable specification of the transition relation, this kind of description can be directly used to interpret DSMs.

By describing operational semantics in terms of the language structure, the domain level of abstraction is kept so that domain experts can understand them. Thus, they can assess the operational semantics of a DSML by observing the execution of example DSMs. This allows to integrate them tightly into the prototyping process of DSMLs.

1.3 Debugging Executable Models

During execution, domain experts might observe erroneous behavior of models. The cause can lie inside the example DSM or in the prototypical semantics description of the DSML. To identify and correct such errors, domain experts and language engineers need debugging support.

Debugging means to control the execution process and to access and possibly modify the runtime state. Common features of debuggers are stepwise execution of programs and setting of breakpoints or, more generally, suspending and resuming the execution at designated execution states. Once the execution is halted, variable values can be inspected and modified. This type of debugging support for DSMLs can easily be achieved when using model-based operational semantics (Sadilek and Wachsmuth 2008b).

1.4 Undoing Model Execution

Based on these foundations for model execution and debugging, in this paper, we want to go further in providing debugging support.

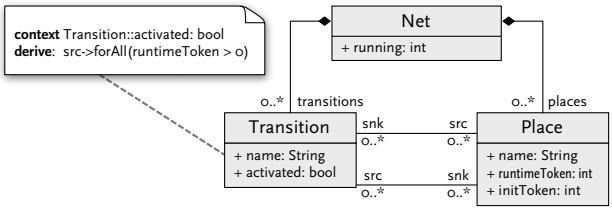


Figure 1. A Petri net metamodel with attributes for storing runtime states.

During DSML prototyping, we encountered the following problem. Assume, for example, a complex model and a configuration that can be reached only by a long series of execution steps that require user input (e.g., in the form of input data or by controlling the behavior of non-deterministic language concepts). Now assume that a domain expert finds an error in this configuration. The execution can be halted and the operational semantics can be changed by a language engineer. Afterwards, the execution has to be restarted and the whole series of execution steps, including every user input, has to be repeated.

To save this work, we propose to allow users to undo execution steps performed by the transition transformation. Thus, the configuration before the error occurred can be restored, and the execution can be resumed. The user can instantly review the effects of the changed operational semantics.

1.5 Structure of the Paper

The rest of this paper is structured as follows. In the following section, we give an example for animated model execution and identify the requirement to undo operational steps of DSMLs. In Sec. 3, we present our approach for undoing operational semantics, and in Sec. 4, we give a brief overview of our implementation. We discuss existing work about DSM execution in Sec. 5 and conclude with our contribution and future work in Sec. 6.

2. Animated Execution and Debugging of Petri Nets

In this section, we give an example of animated model execution and identify the requirement to undo operational steps of DSMLs. We use Petri nets as an example DSML and specify its operational semantics in Java. As execution engine, we use the tool EPROVIDE, which our implementation will eventually be based on.

2.1 Animated Execution and Debugging with EProvide

In (Sadilek and Wachsmuth 2008b), operational semantics are combined with existing editor generation technology in order to support rapid prototyping of visual interpreters and debuggers. The runtime state of a DSM, its configuration, is represented as a model (as we described in Sec. 1.2). A graphical editor that is specific to the configuration metamodel is used to display and modify configurations. By visualizing successive configurations, the editor animates model execution. This approach is implemented in the Eclipse-based tool EPROVIDE, which allows the use of different description languages (Java, QVT Relations, Scheme, Prolog, and Abstract State Machines) to describe operational semantics of metamodel-based languages (Sadilek and Wachsmuth 2008a).

2.2 Describing the Operational Semantics of Petri Nets

As an example, we describe operational semantics of Petri nets. Figure 1 shows a metamodel for Petri nets and their runtime configurations.[1] A Petri net consists of an arbitrary number of places and transitions. Places and transitions can have names and places are marked with a number of tokens. We distinguish the initial marking (`initToken`) and the runtime marking (`runtimeToken`) of places. The initial marking of a place is a "static" attribute that does not change when the model gets executed. The runtime marking is a "dynamic" attribute that encodes the runtime configuration and that changes during model execution.[2] When a model is reset to its initial state, the runtime marking of each place is set to its initial marking. Whether `initToken` or `runtimeToken` is displayed by the editor depends on the value of the Net's (`running`) attribute. Transitions have input (`src`) and output places (`snk`). Depending on its input places, a transition might be activated or not (`activated`).

```
public class PetriSemantics implements ISemanticsProvider {
    public void step(Resource model) {
        Net net = (Net) model.getContents().get(0);
        net.setRunning(true);
        fireTransition(net);
    }

    protected void fireTransition(Net net) {
        EList<Transition> ats = findActivatedTransitions(net);
        if (!ats.isEmpty()) {
            Transition t = choose(ats);

            Place p = choose(t.getSrc());
            consume(p);

            p = choose(t.getSnk());
            produce(p);
        }
    }

    protected T choose(List<T> list) {
        // Returns a randomly chosen member of list.
    }
// ...
}
```

Figure 2. Operational semantics for Petri nets described with Java.

Figure 2 shows a class implementing (erroneous) operational semantics for Petri nets in Java. The method `step()` implements the transition transformation. It is called repeatedly by EPROVIDE to perform operational steps. The API used in `step()` is generated by the Eclipse Modeling Framework (EMF), which we use in our implementation (Sec. 4).

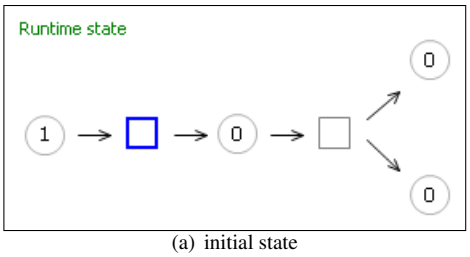
2.3 Animated Execution of a Petri Net

Figure 3 shows what animated execution with EPROVIDE looks like. We use the standard notation for Petri nets: places are represented by circles, transitions by squares; the number of tokens on a place is shown inside the circle. Figure 3(a) shows the initial state of an example model. In Fig. 3(b), the first transition has fired and the token is now on the middle place. Fig. 3(c) shows the state after the second transition fired.

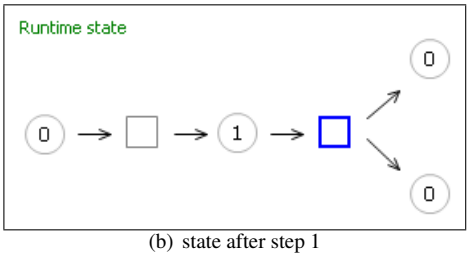
The final state in Fig. 3(c) contains an error that is easily identified by a domain expert. A token was produced at only one of the two output places, but a token should have been produced on all output places. Such errors can be caused by misunderstandings be-

[1] In general, one could use a separate metamodel for the runtime configurations. But to keep this example simple, we use an integrated metamodel.

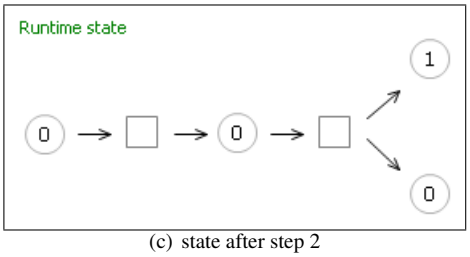
[2] In more complex cases, it is often not sufficient to add attributes to existing metamodel classes; instead, additional classes are necessary to model runtime configurations. This is the case, for example, for reentrant functions, which require that values of local variables are stored for each function call.



(a) initial state



(b) state after step 1



(c) state after step 2

Figure 3. Execution sequence of a Petri net model using erroneous operational semantics.

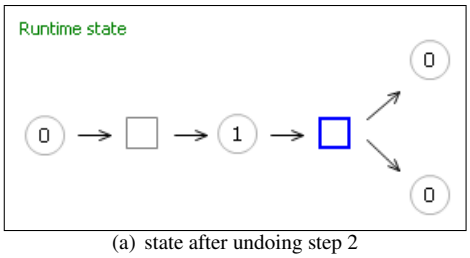
```
// ...
protected void fireTransition(Net net) {
    EList<Transition> ats = findActivatedTransitions(net);
    if (!ats.isEmpty()) {
        Transition t = choose(ats);

        for (Place p : t.getSrc())
            consume(p);

        for (Place p : t.getSnk())
            produce(p);
    }
}
// ...
```

Figure 4. Corrected operational semantics for Petri nets.

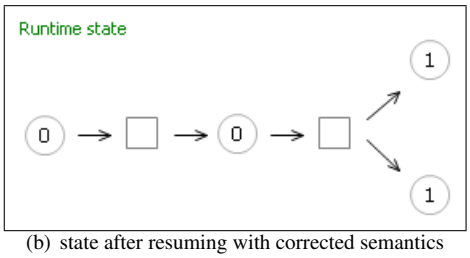
tween domain experts and language engineers. Here, the language engineer has assumed that exactly one token is transfered each time a transition fires. After finding the error by animated execution, the domain expert can explain to the language engineer that the correct behavior for a transition would be to consume one token from all its input places and produce one on all its output places. Now, the language engineer can correct the operational semantics. The corrected code is shown in Fig. 4.

2.4 Requirement to Undo Operational Steps

Now, the capability to undo model execution steps would come in handy. Via undo, the state before the last (incorrect) step could be reached, shown in Fig. 5(a), which is identical to the state shown in Fig. 3(b). From this point, the execution could be resumed, using



(a) state after undoing step 2



(b) state after resuming with corrected semantics

Figure 5. Execution sequence of the Petri net model after correcting the operational semantics.

the corrected operational semantics, which would lead to the state shown in Fig. 5(b).

To control such an undo feature and animated model execution in general, the user needs a graphical user interface. IDEs already provide debugging support for standard programming languages, including a debug user interface for stepwise execution. The undo feature for DSMs should integrate seamlessly into such an existing infrastructure.

3. An Approach for Undoing Operational Steps

We want to allow undoing of DSM execution steps. In each of the steps, the configuration is changed. To undo an execution step, the corresponding model changes have to be undone. A similar problem is already solved for model editors. Here, the model changes are not performed by an operational semantics but by a user. To make model changes undoable, they get encapsulated in undoable units of work that are managed on an undo stack. Our approach is to use similar techniques with the goal to reuse as much of an existing model editor implementation as possible.

3.1 Logging Model Changes

To be able to undo an execution step, it is necessary to know all changes that were performed by the transition transformation. Therefore, the execution engine needs to be notified of all model changes, and the changes have to be stored to create an undoable execution history. This notification problem does not occur with a model editor because the editor is implemented to encapsulate all model changes in undoable units of work. In contrast, with model-based operational semantics (Sec. 1.2), the semantics description does not contain such an encapsulation.

How notification of changes can be achieved is implementation dependent. Two possible ways are: (1) the operational semantics is not applied to the model directly but to a proxy that logs all model changes; (2) the observer pattern is used to receive and log notifications of model changes.

3.2 Integrating Model Execution and Editing

Another issue, which is related to the visualization approach introduced in Sec. 2.1, is the synchronization of editor and execution

engine. The execution engine works on an instance of the configuration metamodel, transforming this instance with every execution step. The editor needs to work with the same configuration to visualize the current runtime state. Editor and execution engine access the configuration concurrently. This can cause inconsistent states. For example, the user could modify a value that the engine just read and still relies on during an execution step. To avoid such interferences, it is necessary to synchronize write access of editor and engine.

3.3 Composing Execution Steps

There is an important difference between changes made in an editor and those made by the execution engine. Generally, the user manipulates one model element at a time with an editor. But a single execution step will, in most cases, include a series of elementary changes of a configuration. Even in the simple Petri net example (Sec. 2), one step, i.e. firing a transition, comprises consuming one token from all input places and producing a token on all output places. All those changes have to be stored together and they must be associated with the execution step they belong to. Furthermore, either the complete step has to be performed or none of the elementary changes. Regarding the synchronization problem from the previous section, this means that the mutual exclusion of editor and execution engine has to span the whole execution step, not just elementary changes. Those requirements, namely atomicity and isolation, are met by transactions. Therefore, all changes belonging to one step must be wrapped into a single unit of work that is executed inside the scope of a transaction.

3.4 Managing Execution Steps

As we want to be able to undo more than one execution step, we have to manage the execution history on a stack. This is straightforward when taken for granted that one step is a single unit of work.

However, a consistency problem arises because the user can modify configurations with the editor. Assume a set M of possible configurations, a set C of possible model changes, and a transition transformation $\rightsquigarrow$: $M \rightarrow M \times C$. Let $m_1 \in M$ be an arbitrary configuration. Applying $\rightsquigarrow$ to m_1 results in a new configuration $m_2 \in M$ with the change c: $m_1 \overset{c}{\rightsquigarrow} m_2$. We can undo c and get back from m_2 to m_1 by reversing the transformation: $m_2 \overset{c^{-1}}{\rightsquigarrow} m_1$. If, however, the user changed m_2 to m_3 using the model editor, we cannot safely undo c in the current state m_3 anymore. That is because the user made the change $m_2 \overset{d}{\rightsquigarrow} m_3$ and generally, $c \neq d$. Therefore, undoing c in state m_3 could lead to an inconsistent state, which may even violate constraints of the metamodel.

To solve this problem, all model changes, regardless of their origin, have to be kept and undone in the order in which they were applied. This can be achieved by using the same stack for both editor and execution engine.

4. Implementation

This section sketches the implementation of our approach, which is based on the tool EPROVIDE[3], introduced in Sec. 2.1.

4.1 Implementation Basis

Important for our implementation are especially the Eclipse Modeling Framework (EMF), the Graphical Modeling Framework (GMF) and Eclipse's debug infrastructure. EMF uses Ecore as meta-metamodel, which is virtually an implementation of OMG's Essential MOF. EMF can generate a Java API for an Ecore metamodel. This API can be used to access metamodel instances programmatically. The Java-based semantics for Petri nets from Sec. 2.2 uses such an API. EMF uses `Resources`, which are an abstraction from physical files, to handle metamodel instances. Resources can reference, or be related to, other resources. They are managed in `ResourceSets`, which in turn are managed by `EditingDomains`. GMF allows graphical editors to be generated for EMF-based models from declarative descriptions. In EPROVIDE, such editors are used to visualize configurations (Sec. 2.1). Eclipse's debug infrastructure provides a language independent debug model and a user interface for common debug functionality.

4.2 Logging Model Changes

In Sec. 3.1, we presented two ways to get notified of changes applied to configurations. The second of them, the observer pattern, is already implemented in EMF: observed model elements are called notifiers and there is already a recording observer called `ChangeRecorder`. Consequently, we chose the observer pattern to get notified of configuration changes. EMF also provides a data structure to store changes, additions, and deletions of model elements: the `ChangeDescription`. All changes belonging to one execution step can be stored in one `ChangeDescription` by using a `ChangeRecorder`. The changes in a `ChangeDescription` can be undone by applying them in reverse to a configuration. Thus, execution steps can be undone.

4.3 Integrating Model Execution and Editing

Following our approach for undoing operational semantics, the EPROVIDE execution engine and the GMF generated editor have to share access to configurations (as was explained in Sec. 3.2). Configurations are EMF models, which are handled in the form of EMF `Resources`. EMF `Resources` can be shared, e.g., between different editors (Wegert and Shatalin 2008). This is achieved by using a shared `EditingDomain`. Following this principle, we use the same `EditingDomain` for editor and execution engine. This grants some additional benefits as we will see in Sec. 4.5.

To solve the problem of synchronizing editor and execution engine, we use EMF's transaction framework EMFT. EMFT is an extension of EMF that provides transactional access to EMF resources. GMF generated editors and EPROVIDE both use EMFT. Thus, synchronization is achieved.

4.4 Composing Execution Steps

As stated in Sec. 3.3, the elementary changes performed during an execution step have to be wrapped into a single unit of work. For this, we use EMFT's `RecordingCommand`, which internally uses a `ChangeRecorder`. For each execution step, a `RecordingCommand` is used that creates a transaction context for the configuration changes in that step. The `RecordingCommand`'s `ChangeRecorder` tracks all configuration changes of the step and stores them in a `ChangeDescription` so that the `RecordingCommand` can be undone. Thus, all model changes of one execution step can be undone as a whole.

4.5 Managing Execution Steps

The shared `EditingDomain`, introduced in Sec. 4.3, also yields a shared `CommandStack`. The `RecordingCommands` from the execution engine and the commands from the editor are now executed and managed via this `CommandStack`. By this means, all commands are stored in the sequential order that was required in Sec. 3.4.

4.6 Integrating the User Interface

Eclipse's debug infrastructure can be extended with new debug functionality. We added support for stepwise DSM execution and

[3] EPROVIDE is open source software and available for download at `http://eprovide.sf.net`

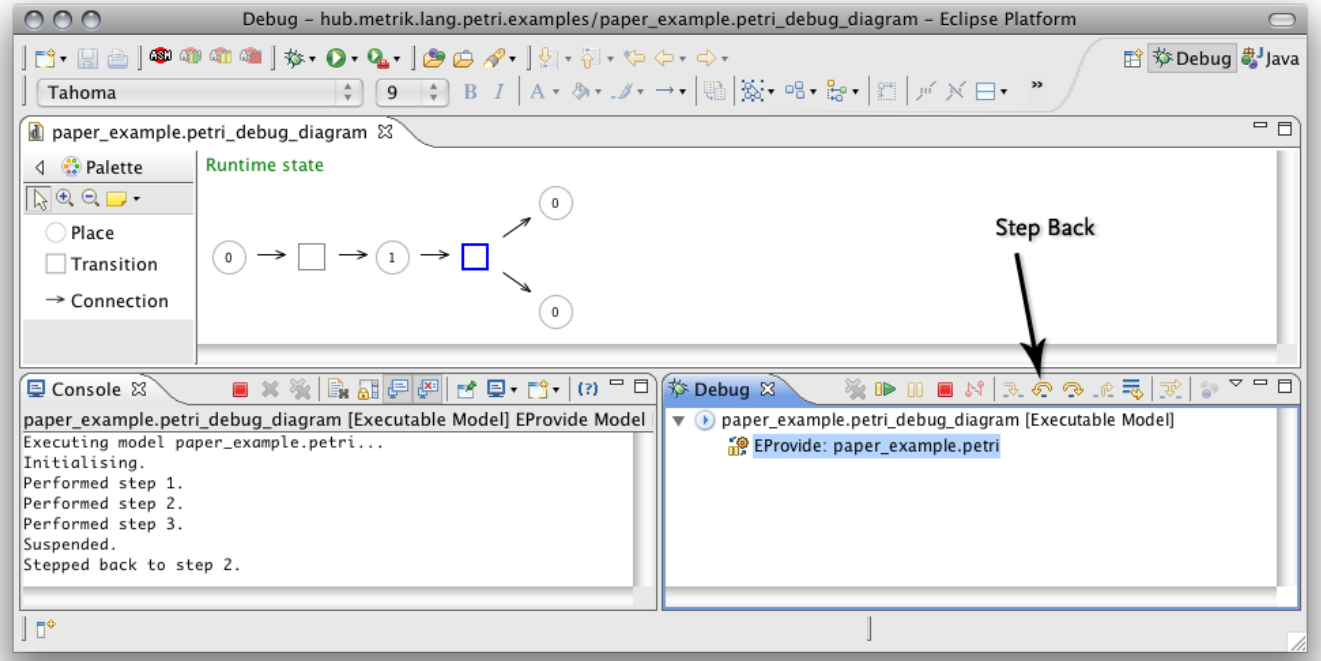


Figure 6. Model debugging with EPROVIDE. The debug toolbar is extended with a button for stepping back in the execution history.

for stepping back in the resulting execution history. The Eclipse debug user interface already provides buttons to suspend, resume, terminate, and stepwise execute a program. We reuse them for model execution. For stepping back in the execution history, we added a "step back" button. Fig. 6 shows a screenshot of EPROVIDE with the extended debug interface.

5. Related Work

Runtime States as Models. Several approaches are based on metamodeling runtime states. All of them provide their own description languages, sometimes a variation or extension of existing modeling languages (Scheidgen and Fischer 2007), OCL (Muller et al. 2005; Clark et al. 2004) or Abstract State Machines (ASMs) (Di Ruscio et al. 2006). None of these approaches has support for stepping back in the execution history. Adding this feature would require solving the following problem: With EPROVIDE, the execution of the transition transformation is stateless and the complete runtime state is encoded in a model. The approaches above, in contrast, use description languages whose execution is stateful, i.e., between execution steps, a state is held inside the interpreter of the description language. For example, with the OCL-based, imperative description language of Muller et al. (2005), the interpreter state is a call stack with the local variables of procedure calls. Therefore, undoing an execution step in these approaches would require not only to undo the changes of the configuration but also to undo the changes of the description language's interpreter state.

Runtime States as Attributed Graphs. Graph transformations are a well-known technology to describe the operational semantics of visual languages (Engels et al. 2000; Ermel et al. 2005). In AToM3, de Lara and Vangheluwe (2004) use graph grammars to define the operational semantics of a visual modeling language. The Moses tool suite (Robert Esser 2001) provides a generic architecture to animate visual models, with execution semantics of models given as ASMs. As they are, these tools do not support stepping back in the execution history. But our approach to reuse model editing techniques could be applied to them, as well.

Hand-Crafted Interpreters. Interpreters for DSMLs can also be implemented manually. Ptolemy (Brooks et al. 2007) allows animated execution of hierarchically composed domain-specific models with different execution semantics. Adding a new DSML to Ptolemy requires that its syntax and its semantics are coded manually in Java. GME (Lédeczi et al. 2001) provides visualization of model interpretation and support for creating a DSML editor without manual coding. But as with Ptolemy, the interpreter semantics has to be implemented manually in Java or C++. The runtime states in these frameworks are encoded in data structures of the language used to implement the interpreter (Java, C++). In contrast to EMF, there is no editing framework with support for undoable commands already available. Therefore, if stepping back in the execution history is to be supported for manually implemented DSML interpreters, using a proxy-based approach like that sketched in Sec. 3.1 seems reasonable.

Animated Model Execution with Translational Semantics. Animated model execution can be achieved not only with an operational but also with a translational semantics description. For this, the DSM editor must provide an API with callback functions that are called by the generated code to reflect the current runtime state in the editor. This approach is supported, for example, by the tool MetaEdit+ (Tolvanen et al. 2007). Using this approach, an undo feature for execution steps cannot be provided generically. Whether and how execution steps can be undone is specific to the platform the generated code runs on.

6. Conclusion

Contribution. In this paper, we showed how prototyping of DSMLs with operational semantics can be improved by support for stepping back in the execution history of DSMs. We presented

an approach for this support that is based on reusing model management techniques normally used in model editors. We proved the feasibility of our approach with an implementation based on the tool EPROVIDE.

Future Work. Execution control, such as support for breakpoints between DSL execution steps, can be achieved by extending the configuration metamodel with elements for controlling the execution process and by adapting the transition transformation to use these elements (Sadilek and Wachsmuth 2008b). For Petri Nets, this would mean to suspend the execution if a certain number of tokens on a specific place is reached or if a selected transition gets activated or deactivated. But at the moment, a user interface for managing breakpoints has to be programmed manually for each DSML. We want to investigate whether breakpoint support for DSMLs can be described declaratively.

Another common debug feature is source lookup. For model debugging, this means accessing and showing those parts of the operational semantics that are related to selected model elements. EPROVIDE is extensible to allow operational semantics descriptions in different languages. We want to investigate if and how generic source lookup for different semantics description languages can be provided.

Some debugging platforms include possibilities to change the structure of a debugged program at runtime. This is done, to some degree, with Java hot-code-replacement. In EPROVIDE, an editor is used for configuration visualization that allows a user to make arbitrary changes of configurations. To prevent a user from producing invalid configurations, the possible changes would have to be constrained. We want to investigate how such constraints can be described and implemented.

When a user finds an error in the operational semantics during execution, he can correct it and undo the step(s) in which he noticed the error (as described in Sec. 2). But maybe the error already occurred earlier and the user just did not notice. In this case, the current runtime state would have never been reached with the corrected operational semantics. EPROVIDE does not deal with this issue. To do so, it would be necessary to trace which parts of the operational semantics description are applied at which execution steps. This would enable EPROVIDE to automatically step back to the last state in the execution history that is consistent with the changed semantics. In principle, such a feature is possible (e.g., it was implemented for the visual, functional language Prograph (Cox and Pietrzykowski 1985)) but its feasibility strongly depends on the used description language.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work is partially supported by grants from the Deutsche Forschungsgemeinschaft, Graduiertenkolleg METRIK (GRK 1324).

References

Christopher Brooks, Edward A. Lee, Xiaojun Liu, Steve Neuendorffer, Yang Zhao, and Haiyang Zheng. *Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java*. UC Berkeley, 2007.

T. Clark, A. Evans, P. Sammut, and J. Willans. *Applied metamodelling: A foundation for language driven development*. `www.xactium.com`, 2004.

P. T. Cox and T. Pietrzykowski. Advanced programming aids in prograph. In *SIGSMALL '85: Proceedings of the 1985 ACM SIGSMALL symposium on Small systems*, pages 27–33, New York, NY, USA, 1985. ACM.

Juan de Lara and Hans Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. *Journal of Visual Languages & Computing*, 15(3-4):309–330, 2004.

Davide Di Ruscio, Frederic Jouault, Ivan Kurtev, Jean Bezivin, and Alfonso Pierantonio. Extending amma for supporting dynamic semantics specifications of dsls. Technical Report HAL - CCSd - CNRS, Laboratoire D'Informatique de Nantes-Atlantique, 2006.

G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic metamodeling: A graphical approach to the operational semantics of behavioral diagrams in uml. In *UML'00*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.

Claudia Ermel, Karsten Hölscher, Sabine Kuske, and Paul Ziemann. Animated simulation of integrated uml behavioral models based on graph transformation. In *VL/HCC'05*, pages 125–133. IEEE Computer Society, 2005.

Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001. ISSN 0018-9162.

P.A. Muller, F. Fleurey, and J.M. Jézéquel. Weaving executability into object-oriented meta-languages. In *MoDELS'05*, volume 3713 of *LNCS*, pages 264–278. Springer, 2005.

Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

Jorn Janneck Robert Esser. Moses: A tool suite for visual modeling of discrete-event systems. In *HCC'01*, pages 272–279. IEEE Computer Society, 2001.

Daniel A. Sadilek and Guido Wachsmuth. Eprovide 2.0: Where grammarware meets modelware. Technical report, Humboldt-Universität zu Berlin, 2008a.

Daniel A. Sadilek and Guido Wachsmuth. Prototyping visual interpreters and debuggers for domain-specific modelling languages. In Ina Schieferdecker and Alan Hartman, editors, *ECMDA-FA 2008*, volume 5095 of *LNCS*, pages 63–78, Berlin, Germany, 2008b. Springer.

Markus Scheidgen and Joachim Fischer. Human comprehensible and machine processable specifications of operational semantics. In *ECMDA'07*, volume 4530 of *LNCS*, pages 157–171. Springer, 2007.

Juha-Pekka Tolvanen, Risto Pohjonen, and Steven Kelly. Advanced tooling for domain-specific modeling: MetaEdit+. In J. Sprinkle, J. Gray, M. Rossi, and J.-P. Tolvanen, editors, *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07)*, number TR-38 in Computer Science and Information System Reports. University of Jyväskylä, 2007.

Guido Wachsmuth. Modelling the operational semantics of domain-specific modelling languages. In *GTTSE 2007*, LNCS. Springer, 2008. to appear.

Volker Wegert and Alex Shatalin. *Integrating EMF and GMF Generated Editors*, March 2008. URL `http://www.eclipse.org/articles/article.php?file=Article-Integrating-EMF-GMF-Editors/index.html`.

The Interchange of (Meta)Models between MetaEdit+ and Eclipse EMF Using M3-Level-Based Bridges

Heiko Kern

Business Information Systems, University of Leipzig
Johannisgasse 26, 04103 Leipzig, Germany
kern@informatik.uni-leipzig.de

Abstract

Nowadays there are powerful tools for Domain-Specific Modeling. An ongoing problem is the insufficient tool interoperability which complicates the development of complete tool chains or the re-use of existing metamodels, models, and model operations.

In this paper we present the approach of M3-Level-Based Bridges and apply this approach to enable the interoperability between two selected tools. The first tool is MetaEdit+ with strengths in (meta)modeling and the second tool is the Eclipse Modeling Framework with advantages in model processing by transformation and generation tools.

General Terms Model-Driven Engineering, Metametamodel, Domain-Specific Language, Interoperability

Keywords Tool Interoperability, MetaEdit+, Eclipse Modeling Framework, Model Transformation, M3-Level-Based Bridge

1. Introduction

Domain-Specific Modeling (DSM) is a software development approach which basically uses the following two concepts. First, DSM applies special-purpose languages covering the domain concepts of the problem space. Second, DSM uses model operations such as generators or transformations working on domain-specific models to automate the development of executable software (Kelly and Tolvanen 2008). Similar to other development approaches, the tool support is a crucial factor for the success of DSM. Fortunately, powerful tools are available that support typical DSM tasks such as language definition, modeling, and model processing. Some of these tools are, the Microsoft DSL Tools (Cook et al. 2007), MetaEdit+ (MetaCase 2008), Generic Modeling Environment (Ledeczi et al. 2001), Eclipse Graphical Modeling Framework (GMF 2008), openArchitectureWare (oAW 2008) or the AMMA Platform (AMM 2008).

But an ongoing problem is the insufficient tool interoperability. This lack of interoperability complicates the development of complete tool chains or the re-use of existing metamodels, models, and model operations (Bézivin et al. 2005a; Karsai et al. 2003). In this article, we present the approach of M3-Level-Based Bridges to achieve tool interoperability between different DSM tools. We apply this approach to realize the interchange of metamodels and models between two selected tools: MetaEdit+ (abbreviated MetaEdit) and the Eclipse Modeling Framework (abbreviated Eclipse EMF).

MetaEdit from MetaCase is a widely-used commercial DSM tool which supports the developer during the definition of Domain-Specific Languages (DSLs) and allows modeling with these DSLs. Further, the tool includes a code generator. In addition to the DSM functionality, MetaEdit provides an extensive model repository with multi-user functionality.

The second tool that supports typical DSM tasks is given by the Eclipse Modeling Framework and by tools based on this framework. Eclipse EMF allows the definition of domain-specific metamodels and provides basic functionality such as an API to access (meta)models, a serialization function to XMI/XML, and a generator language. Based on Eclipse EMF, the selection of tools is diverse, ranging from model transformation tools (Jouault and Kurtev 2005; Lawley and Steel 2005)and tools for DSL modeling (GMF 2008; Grundy et al. 2008)to other model processing tools (Fabro et al. 2005; EMF 2008).

The approach of M3-Level-Based Bridges will be applied on MetaEdit and Eclipse EMF because both tool spaces can benefit from the created interoperability. While MetaEdit has advantages in (meta)modeling, Eclipse EMF provides a wide range of model processing tools.

The paper is structured as follows: in the subsequent section, we will give a conceptual overview of M3-Level-Based Bridges. As the bridge is based on the mapping between metametamodels, we will explore the metamodeling language of MetaEdit and the metametamodel of Eclipse EMF in section 3.1 and 3.2, respectively. In section 4 we will present the development of the bridge. In section 5 we will describe the implementation of the bridge and will show an example in section 6. Lastly, we will summarize the article and will conclude with future challenges.

2. M3-Level-Based Bridges

2.1 Conceptual Overview

The idea of M3-Level-Based Bridges is to achieve interoperability between different tools by transforming models and metamodels. This approach is well-established and has been successfully applied in building bridges between MetaGME and Eclipse EMF (Bézivin et al. 2005b), Microsoft DSL Tools and Eclipse EMF (Bézivin et al. 2005c), ARIS Toolset and Eclipse EMF (Kern and Kühne 2007), or Meta Object Facility and Eclipse EMF (Duddy et al. 2003). Although the implementation of such bridges can differ in technical terms, the conceptual approach is the same.

A prerequisite to construct bridges is the existence of a metamodel hierarchy (Kühne 2006; Atkinson and Kühne 2003; Gitzel and Hildenbrand 2005) consisting of three levels. At the lowest level (M1-level) are models which describe a software system. The structure of these models and the

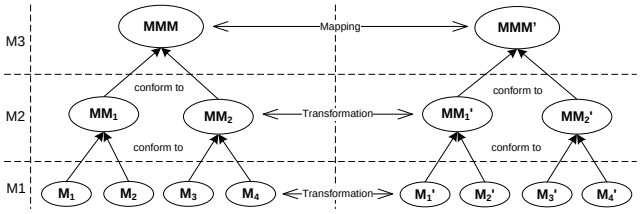


Figure 1. Conceptual Approach of M3-Level-Based Bridges.

available concepts that can be instantiated in models are defined by a metamodel at M2-level. Finally the structure and the available concepts of the metamodels are defined by a metametamodel at M3-level. Such a M3-level hierarchy often occurs in DSM tools because it easily enables the development of DSLs.

Based on the existence of such hierarchies, the basic step to build M3-Level-Based Bridges is the mapping between the metametamodels (see Fig. 1). The mapping consists of different mapping rules specifying the relation between semantically equivalent concepts. Semantically equivalent means, for instance, that concepts at M3-level expressing relationships at M2-level are mapped onto each other. Based on the mapping specification the transformation of metamodels and models can be derived. To create the transformations, it is necessary to know how the instance relationship is realized between each level. The M2-level transformation maps metamodels between hierarchies. These metamodels are isomorphic. Analogous to the M2-Level transformation, the M1-level transformation enables the mapping of models. These models are also isomorphic.

2.2 Typical Metamodeling Concepts

The M3-level mapping is the basic concept of M3-Level-Based Bridges. Many metametamodels have similar concepts for metamodeling that can be mapped on each other. In this section we want to describe these typical metamodeling concepts (i. e. a kind of equivalence classes for metamodeling concepts) to provide an assistance for the latter M3-level mapping between MetaEdit and Eclipse EMF (in Sec. 4.2). Moreover, these general concepts can be helpful to build other M3-level mappings.

We have analyzed the following metametamodels: Ecore from Eclipse EMF (Budinsky et al. 2004), GOPRR from MetaEdit (Kelly and Tolvanen 2008; Kelly 1997; Tolvanen 1998), A3 from ARIS Toolset (Kern and Kühne 2007), MOF (version 1.4) (Obj 2002), and the metametamodel from Microsoft DSL Tools (Bézivin et al. 2005c; Cook et al. 2007). As a result, we suggest the following classes of metamodeling concepts:

Object type An object type is a concept to define a set or a class of objects with equal features. Other names for object type can be class, metaclass, entity, or object.

Relation type A relation type defines a set of relations between objects. Other notations are relationship, reference, connection, or association.

Attribute type Attribute types define features for metamodel elements such as object types or relation types. Another name can be property type.

Data type A data type specifies the range of values in attributes. Typical data types are, for instance, `integer`, `string`, or `date`. But it can also be object types or relationship types.

Model type A model type is a concept to define a set of models consisting of defined metamodel elements. Other notations are graph type, domain model or metamodel.

Inheritance This concept can be used to relate metamodel elements in a inheritance relationship. Usually this concept can be interpreted like the inheritance in object orientation programming languages.

Partitioning This concept allows the (logical) structuring of metamodels in defined parts. Other names can be namespace or package.

Depositation A depositation often specifies a relation between metamodel elements (such as object type or relation type) and model types. Other designations are explosion, decomposition or assignment.

Constraint Constraints enable the specification of conditions which have to be fulfilled during or after modeling.

The results of the study are intentionally abstract because a detailed metametamodel comparison is not the focus of this paper.

3. (Meta)Modeling with MetaEdit and Eclipse EMF

3.1 Domain-Specfic Modeling in MetaEdit

MetaEdit is an established DSM environment which provides a powerful language for metamodeling. The concepts of this (meta)language are defined in the GOPRR model. GOPRR is the abbreviation for Graph, Object, Property, Role and Relationship and is shown in Figure 2.

A graph type (instance of `Graph`) specifies a model type that contains the modeling concepts: object types (instances of `Object`), relationship types (instances of `Relationship`), and role types (instances of `Role`). An object type describes a class of model elements which can exist on their own. Object types can be connected by relationship types, whereby the role type specifies how an object type participates in a relationship type. Relations can be defined by the concept of `Binding`. A binding connects a relationship type, two or more role types, and one or more object types for each role type in a graph type. Each previously described language concept, except for the `Binding` concept, can have property types (instances of `Property`) to characterize language concepts. The values of property types can be different data types such as a string, text, number, boolean, collection, or a link to other modeling language concepts. Other concepts supported in GOPRR are `Inheritance` that makes the creation of subtypes possible, `Decomposition` which enables object types to have subgraph types, and `Explosion` which allows object types, relationship types, or role types to be linked to other graph types.

The notation or concrete syntax of a modeling language can be defined with the help of a symbol editor. The editor supports the creation of symbols for object types, relationship types, and role types. It is possible to place a textual representation of property types in symbols.

After describing the language definition, we need to know the structure of the model repository in order to build the bridge. "GOPRR has been designed to be applicable in the

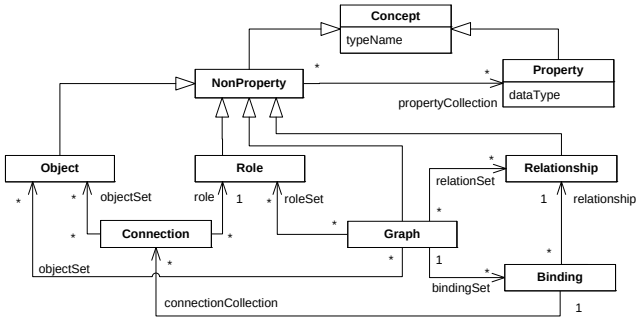


Figure 2. Snippet of GOPRR (Kelly 1997).

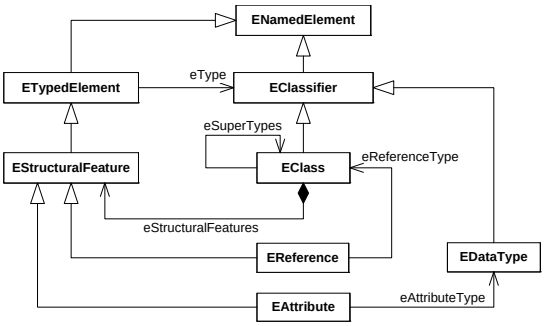


Figure 3. Snippet of Ecore.

same way on both the type and instance levels [...]" (Kelly 1997). That is, the model repository (instance level) is almost equal with the GOPRR model shown in Figure 2. A graph (instance of a graph type) owns a set of relationships (instances of a relationship type), objects (instances of an object type), roles (instances of a role type), and bindings (instances of a binding). A binding stores the combination of a relationship connected with objects and roles. Furthermore, all property values are stored in a value field of the property (instance of property type). All model and metamodel elements are stored in projects. Thus, the GOPRR model contains a project class referencing a graph set.

3.2 Metamodels and Models in Eclipse EMF

Contrary to MetaEdit, developed especially for DSM, Eclipse EMF is designed to support the development of (Eclipse) applications. One reason for this statement is the missing of special metamodeling concepts such as `Model type` or `Depositation` in Ecore. Models (data models) describing Eclipse EMF applications can be regarded as metamodels and instance data of these metamodels as models.

A simplified subset of Ecore is shown in Figure 3 and a detailed description of Ecore is given in Budinsky et al. (2004). The main elements of Ecore are `EClass`, `EReference` and `EAttribute`. An EClass (instance of `EClass`) defines an EMF metamodel element that represents a set of similar model entities. EClasses can have EReferences (instances of `EReference`) which express unidirectional relationships between two EClasses. An EClass can additionally have EAttributes (instances of `EAttribute`) to express properties of the EClass. The range of the attribute values are specified by a data type such as int, string or date. A further metamodeling concept is the inheritance between EClasses.

Analogous to the structure of the MetaEdit model repository, we need to know the structure of the model repository in order to build the M1-level transformation. Every model element in Eclipse EMF is an EObject specified by a Java interface. The EObject interface provides different methods which enable the navigation in models and allow the query of metatype information. For instance, the EObject.eGet(EStructuralFeature) method returns either all EObjects referenced by a certain EReference or values of the EAttributes. Further, the EObject.eClass() method returns the EClass of a model element.

4. MetaEdit to Eclipse EMF Bridge

4.1 Overview of the Bridge

MetaEdit and Eclipse EMF can be structured into three levels: M3 (GOPRR and Ecore), M2 (GOPRR metamodels and EMF metamodels), and M1 (GOPRR models and EMF models). Based on this level structure, we apply the approach of M3-Level-Based Bridge (see Fig. 4).

The M3-level mapping specifies a unidirectional mapping from GOPRR to Ecore. Using this mapping, we can derive the transformation rules at M2-level which export MetaEdit metamodels to Eclipse EMF. Several transformation rules map different GOPRR concepts onto one Ecore concept. To distinguish the different GOPRR concepts in Ecore, we introduce an abstract EMF metamodel (see Fig. 5) that approximates the structure of GOPRR. All exported metamodel elements are inherited from a corresponding abstract metamodel element.

For our purpose, the export of MetaEdit metamodels is sufficient and we do not consider importing EMF metamodels to MetaEdit. But this would also be possible on the condition that all metamodel elements are inherited from an abstract metamodel element.

Based on the M3-level mapping and M2-level transformation, we can derive the M1-level transformation which enables the export and re-import of MetaEdit models.

4.2 M3-Level Mapping

We propose the following mappings:

`Object` ↦ `EClass`**:** In MetaEdit an object type can define a set of model entities. The corresponding concept in Ecore is the `EClass` concept which can also define a set of model entities. Therefore, we map `Object` onto `EClass`. The `typename` of `Object` maps onto `name` of `EClass`.

`Relationship` ↦ `EClass`**:** A relationship type between object types is expressed by `Relationship` in GOPRR. In Ecore the `EReference` concept can be used to describe relationship types between EClasses. The problem is that relationship types in MetaEdit can have property types. But EReferences cannot have their own EAttributes. Hence, we cannot map `Relationship` onto `EReference`, but we can map `Relationship` onto `EClass` whereby `typename` of `Relationship` maps onto `name` of `EClass`.

`Role` ↦ `EClass`**:** A Role type defines how an object type takes part in a relationship type. No direct equivalent concept exists in Ecore. Hence, we map `Role` onto the `EClass` concept. The `typename` of `Role` maps onto `name` of `EClass`.

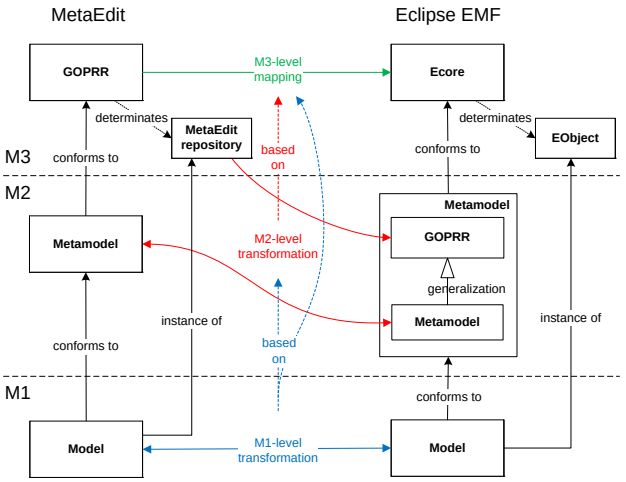


Figure 4. Overview of the MetaEdit to EMF Bridge.

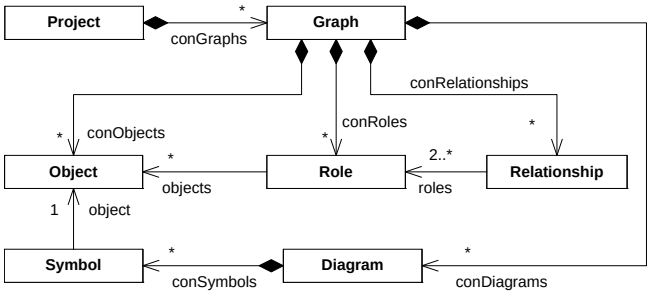


Figure 5. Simplified GOPRR as EMF Metamodel.

`Graph` ↦ `EClass`**:** An equivalent concept for `Graph` does not exist in Ecore. Hence, we map a `Graph` onto `EClass` whereby the `typename` maps onto `name`.

`Property` ↦ {`EAttribute`, `EReference`}**:** We map the `Property` concept onto `EAttribute` or `EReference`. The decision whether it is an `EReference` or an `EAttribute` depends on the data type of a property type. If the data type is a simple type such as string, int or text, then we map `Property` onto `EAttribute`. If the data type is a link to other modeling concepts, then we map `Property` onto `EReference`. The `typename` maps onto `name`.

`Inheritance` ↦ `eSuperType`**:** We map `Inheritance` onto `eSuperType` whereby the related metamodel elements are equal.

4.3 M2-Level Transformation

The M2-level transformation consists of transformation statements which are derived from the M3-level mapping specification. For instance, the mapping rule `Object` ↦ `EClass` transforms all object types in EClasses. These EClasses are inherited from the abstract EClass 'Object' (see Fig. 5) because we need to distinguish those from other EClasses. A further mapping rule `Relationship` ↦ `EClass` transforms all relationship types in EClasses which are all inherited from the abstract EClass 'Relationship'.

4.4 M1-Level Transformation

The M1-level transformation consists of transformation statements which are also derived from the M3-level mapping specification. For instance, the mapping rule `Object` ↦ `EClass` creates the transformation of all MetaEdit objects (instances of a certain object type) to EObjects (instances of the EClass corresponding to the object type). Another example is the rule `Relationship` ↦ `EClass` which creates the transformation of all MetaEdit relationships (instances of a certain relationship type) to EObjects (instances of the EClass corresponding to the relationship typ).

Moreover, model data such as symbol position or diagram name is transformed into EMF models. Hence, the abstract EMF metamodel contains EClasses such as 'Symbol' or 'Diagram'.

5. Implementation of the Bridge

We have prototypically realized the bridge in Java as an Eclipse plug-in. This bridge mainly consists of three parts. The first part is the M2-level transformation which creates EMF metamodels from MetaEdit language definitions. MetaEdit provides an XML export for the language definition. The schema of these XML files is vendor-specific and described in the tool documentation[1]. We have implemented a GOPRR-Reader which can query the XML export and return, for instance, all graph types, object types, and relationship types. Furthermore, we have implemented the transformation rules in Java by using the EMF API. The implementation builds an in-memory object model and serializes the objects as an XMI file in Ecore format.

The second part of the bridge is the M1-level transformation which transforms MetaEdit models to EMF models. MetaEdit provides two possibilities to access model data, (1) an XML format being similar to the metamodel XML format, (2) a Simple Object Access Protocol (SOAP) API. We use the SOAP API because it provides methods such as objectSet() or relationshipSet() in order to query model elements, and methods such as type() to get the corresponding metamodel elements. The transformation navigates through the MetaEdit model and creates EMF model elements, whereby the EMF model element is an instance of the EMF metamodel element corresponding to the MetaEdit metamodel element. Analogous to the M2-level transformation, an EMF model is created in a dynamic way (i.e. in-memory) and is serialized as an XMI file.

The third part of the bridge is the M1-level transformation which transforms EMF models to MetaEdit models. The transformation also uses the EMF API to navigate through the EMF models and creates a corresponding MetaEdit model element for each EMF model element by using the SOAP API.

6. Application of the Bridge

A concrete example will allow to demonstrate how we can reuse validation rules of Event-Driven Process Chains (EPC) (Nüttgens and Rump 2002) which were already implemented as part of a validation approach for business process models (Kühne et al. 2008).

EPC is a graphical modeling language to describe business processes. EPC models consist of nodes and arcs. Nodes can be functions (activities which need to be executed, depicted as rounded boxes), or events (representing pre- and

[1] http://www.metacase.com/support/45/manuals/mwb/Mw.html

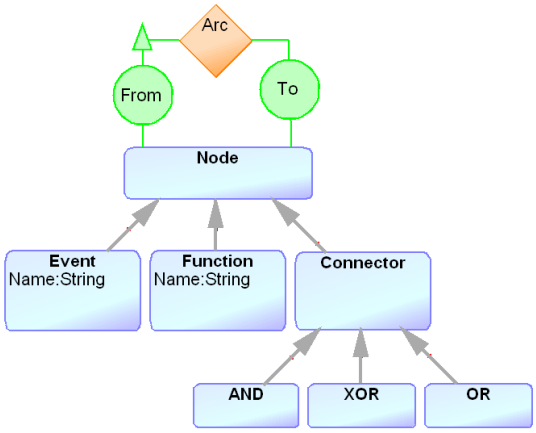


Figure 6. EPC Metamodel in MetaEdit-specific Notation.

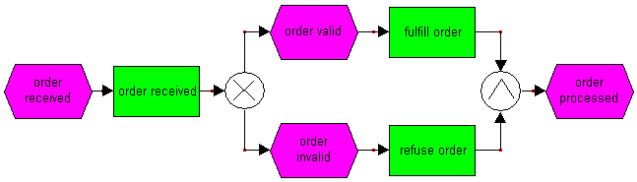


Figure 7. Incorrect EPC Model in MetaEdit.

postconditions of functions, depicted as hexagons), or connectors. Arcs between these elements represent the control flow. Connectors are used to model parallel (AND-connectors ⓐ) and alternative (XOR-connector ⓧ and OR-connector ⓥ) executions. Figure 6 shows the corresponding EPC metamodel in a MetaEdit-specific notation. There are the following objects types: 'Node', 'Event', 'Function', 'Connector', 'AND', 'XOR' and 'OR'. The arc concept is realized by a relationship type 'Arc' which is connected with 'Node' by the role types: 'From' and 'To'. Furthermore, the types 'Event' and 'Function' have a property type 'Name'.

After describing the EPC language definition in MetaEdit, we can create EPC models which are to be checked later. Figure 7 shows such an EPC model which is incorrect because of the mismatch between the type of the first connector and the type of the second connector. This construct will always result in a deadlock because the XOR-split starts only one control flow and the AND-join waits for both flows to be completed.

Now, we can export the metamodel and model from MetaEdit to Eclipse EMF by using the bridge. Afterwards, we need to transform the exported EMF-MetaEdit models to EMF models conforming to an EPC metamodel used by the validation rules. This transformation is easy to realize by a model-to-model transformation in Eclipse EMF.

Thereafter, we can apply the validation rules to check the models. The rules are expressed in the Check language from openArchitectureWare. Listing 1 shows an example rule. A rule is introduced by a context and an optional if-clause specifying a set of model elements that should be validated. The ERROR keyword and the following message signal a violated validation rule. The boolean expression after the colon provides a validation assertion which holds for valid models. In the given case, the rule in Listing 1 detects the mismatch of the connectors from Figure 7 and creates an error message. Currently, the error messages are displayed in the Eclipse workbench but it is also possible to create error messages in the MetaEdit workbench by using the SOAP API.

```
// XOR−AND−Mismatch
context epc::Connector if (this.isAndJoin())
    ERROR "Mismatched XOR−split ..."
    this.allPredessesors().notExists(p| p.isXorSplit()
     && p.seseMatch(this));
```

Listing 1. XCheck Rule: Mismatched XOR-Split/AND-Join

The above example is very specific but it is also possible to use other EMF tools. Furthermore, the usage of other bridges from Eclipse EMF to other metamodel hierarchies enables the usage of further tools. For instance, the Eclipse EMF to Web Ontology Language bridge, implemented in the EODM project[2], enables the application of typical ontologies tools such as reasoner.

7. Summary and Conclusion

In this paper, we developed an interface for the exchange of metamodels and models between MetaEdit and Eclipse EMF by applying the concept of M3-Level-Based Bridges. For this purpose, we explored the MetaEdit language definition concepts and the underlying repository structure. Furthermore, we described the metametamodel Ecore and the generic model storage structure of Eclipse EMF. Based on this information, we specified a M3-level mapping and derived a M2-level and M1-level transformation implemented as an Eclipse plug-in. After, we demonstrated the bridge by an example of EPC validation. The example showed the modeling of EPCs and the application of validation in Check.

M3-Level-Based Bridges have already been used repeatedly to achieve (meta)model interchange between different tools based on metamodel hierarchies. As a result of the development and the study of this bridge and other bridges, we can say that this approach is useful to build tool chains and to re-use models and model operations. But we also identified the following typical problems that need further research:

Expressive power Often, the metametamodels are different, i.e., they provide different metamodeling concepts to express metamodels. For instance, some metametamodels support the `Model type` concept, others do not support this concept natively. Hence, the mapping between metametamodels can be very complex or even impossible.

Synchronization The synchronization of models and metamodels during the exchange is important in real-world use cases. Therefore, we need approaches for traceability links, model differences and model merging.

In the future work, we want to solve the problems mentioned above. Moreover, we want to develop further M3-Level-Based Bridges and want to explicit our experience in terms of a guide which helps with the development of bridges.

[2] http://www.eclipse.org/modeling/mdt/

References

The AMMA Platform. http://www.sciences.univ-nantes.fr/lina/atl/AMMAROOT/, 2008.

Eclipse Modeling Framework Technology (EMFT). http://www.eclipse.org/modeling/emft/, 2008.

Eclipse Graphical Modeling Framework. http://www.eclipse.org/gmf/, 2008.

openArchitectureWare. http://www.openarchitectureware.org/, 2008.

Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, September 2003.

Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. The Eclipse Series. Addison Wesley, 2004. ISBN 01311425420.

Jean Bézivin, Hugo Brunelière, Frédéric Jouault, and Ivan Kurtev. Model Engineering Support for Tool Interoperability. In *Proceedings of the 4th Workshop in Software Model Engineering (WiSME)*, 2005a.

Jean Bézivin, Christian Brunette, Régis Chevrel, Frédéric Jouault, and Ivan Kurtev. Bridging the Generic Modeling Environment (GME) and the Eclipse Modeling Framework (EMF). In *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA'05*, San Diego, California, USA, 2005b.

Jean Bézivin, Guillaume Hillairet, Frédéric Jouault, Ivan Kurtev, and William Piers. Bridging the MS/DSL Tools and the Eclipse Modeling Framework. In *Proceedings of the International Workshop on Software Factories at OOPSLA 2005*, San Diego, California, USA, 2005c.

Steve Cook, Gareth Jones, Stuart Kent, and Alan Cameron Wills. *Domain Specific Development with Visual Studio DSL Tools (Microsoft .Net Development)*. Addison-Wesley Longman, 2007. ISBN 0321398203.

Keith Duddy, Anna Gerber, and Kerry Raymond. Eclipse Modelling Framework (EMF) import/export from MOF / JMI. Technical report, CRC for Enterprise Distributed Systems Technology (DSTC), 2003.

Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, Erwan Breton, and Guillaume Gueltas. AMW: A Generic Model Weaver. In *Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05)*, 2005.

Ralf Gitzel and Tobias Hildenbrand. A Taxonomy of Metamodel Hierarchies. *Working papers / Lehrstuhl für ABWL und Wirtschaftsinformatik*, 2005.

John Grundy, John Hosking, Jun Huh, and Karen Li. Marama: an Eclipse meta-toolset for generating multi-view environments. *IEEE/ACM International Conference on Software Engineering*, 2008.

Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica, 2005.

Gabor Karsai, Andras Lang, and Sandeep Neema. Tool Integration Patterns. *Workshop on Tool Integration in System Development (TIS)*, 2003.

Steven Kelly. *Towards a Comprehensive MetaCASE and CAME Environment: Conceptual, Architectural, Functional and Usability Advances in MetaEdit+*. PhD thesis, University of Jyväskylä, 1997.

Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Son, Inc., 2008. ISBN 0470036664.

Heiko Kern and Stefan Kühne. Model Interchange between ARIS and Eclipse EMF. In Juha-Pekka Tolvanen, Jeff Gray, Matti Rossi, and Jonathan Sprinkle, editors, *7th OOPSLA Workshop on Domain-Specific Modeling at OOPSLA 2007*, 2007.

Stefan Kühne, Heiko Kern, Volker Gruhn, and Ralf Laue. Business Process Modelling with Continuous Validation. In Cesare Pautasso and Jana Köhler, editors, *1st International Workshop on Model-Driven Engineering for Business Process Management*, 2008.

Thomas Kühne. Matters of (Meta-) Modeling. *Software and Systems Modeling*, 5(4):369–385, 2006. doi: 10.1007/s10270-006-0017-9.

Michael Lawley and Jim Steel. Practical Declarative Model Transformation With Tefkat. *Satellite Events at the MoDELS 2005 Conference*, LNCS Vol. 3844, 2005.

Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The Generic Modeling Environment. In *IEEE International Workshop on Intelligent Signal Processing*, 2001.

MetaCase. MetaEdit+. http://www.metacase.com/, 2008.

Markus Nüttgens and Frank J. Rump. Syntax und Semantik Ereignisgesteuerter Prozessketten (EPK). In *Promise 2002 - Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen*, pages 64–77, 2002.

Meta Object Facility (MOF) Specification 1.4. Object Management Group, 2002. URL `http://www.omg.org/docs/omg/03-06-01.pdf`.

Juha-Pekka Tolvanen. *Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence*. PhD thesis, University of Jyväskylä, 1998.

Foundations for a Domain Specific Modeling Language Prototyping Environment

A compositional approach

Luis Pedro
Centre Universitaire D'Informatique,
Université de Genève, Site de Battelle,
Bat. A, Route de Drize, 7, 1227 Carouge,
Switzerland
Luis.Pedro@unige.ch

Didier Buchs
Centre Universitaire D'Informatique,
Université de Genève, Site de Battelle,
Bat. A, Route de Drize, 7, 1227 Carouge,
Switzerland
Didier.Buchs@unige.ch

Vasco Amaral
Faculdade de Ciências e Tecnologia ,
Universidade Nova de Lisboa (UNL),
Lisbon, Portugal
Vasco.Amaral@di.fct.unl.pt

Abstract

Developing in a domain specific environment introduces all the advantages of thinking at the same abstraction level of the problem under consideration. The gap between the real problem and the mental model is reduced with respect to the generic approach of using General Purpose Languages.

In this article we consider that Domain Specific Modeling Languages (DSMLs) can be prototyped using a compositional and incremental approach. We reason over the fact that concepts presented in a DSML can be extended to a more precise semantics and that might be used in different domain environments. The combination of different concepts with an associated semantics allows achieving the desired expressiveness of the DSML.

Keywords Metamodeling, Re-usability, Semantics, Composition, Model Extension, DSML

1. CONTEXTUAL CONSIDERATIONS

The main purpose of Domain Specific Modeling languages (DSMLs) is to allow for domain users to think in terms closer to the problem domain when specifying their systems, by providing a way to model them at the right abstraction level. This approach is pushing for an interesting shift from the traditional programming approach paradigm to a model-specification based one. In fact, the software engineering community agreed that concerning to the rise of accidental complexity of software development (Brooks 1987), the object orientation programming technique has reached its limits (even after the pattern based approach).

As a consequence of all that the design of modeling languages for specific domains is pushing for systematic approaches, techniques and tools to help to drop the complexity of developing DSMLs.

We have recently watched to the rise of language metamodeling as a standard design technique for the purpose of syntax specification together with transformation techniques for mapping the model into well understood formalisms in order to provide semantics.

Although interesting results and techniques have been achieved so far, for the development of individual DSMLs, the community is starting to realize that we are reaching the point were developing a language is still not a trivial task. In fact they are hard to develop, verify or even execute (Ladd and Ramming 1994).

At some point software engineering community started to make use of patterns to help reducing the complexity of a system specification by using the object oriented paradigm. Following the same approach, now at the DSML level, some work has already been done in the direction of finding common language metamodeling patterns (or language "meta-templates") and respective composition. However promising this approach might seem, it only tackles the problem at the syntactic level, leaving out an important part of the complexity of designing a language, the semantics.

Therefore, our major motivation is to deploy a consistent methodology in conjunction with a framework that simplifies the task of specifying semantics to a DSML. This major goal consists in providing functionalities for the tasks of:

a) re-using existing domain concepts for defining families of DSMLs;

b) generating executable prototypes that simulate the DSML behavior;

c) validating and verifying specific and general concepts of the language;

In this paper we will present a conceptual framework that aims to produce prototype generation of DSMLs behavior by using metamodel and transformations' extension and composition.

1.1 Related Work

Previous works (e.g. (Emerson and Sztipanovits 2006)) developed in the area of metamodeling and DSML engineering show that is possible to identify basic patterns that repeat among different DSMLs. These patterns, also known as domain concepts, must be composed for achieving complex structures that can represent the behavior of a domain. The techniques available so far are either tackling the problem purely at the syntactic level (Ledeczi et al. 2001; Vanderbilt University, Institute for Software Integrated Systems 2005), or are too abstract and complex to be used (Jackson and Sztipanovits 2006). In the presented approach we define a methodology for a framework that define a set of semantic mappings, generate simulation traces, executable code, and verification results from models trying to simultaneously use formal constructions to profit from the advantages of having a well defined working environment, and to have an engineering "by construction" approach in order to provide the methodology with a pragmatic ground.

1.1.1 Domain Concept

The fundamental idea that we want to empathize in this paper is the notion of *domain concept* that underlies and influences the approach presented: rather than a metamodel, a *domain concept* is a metamodel that has attached to it transformation to a target

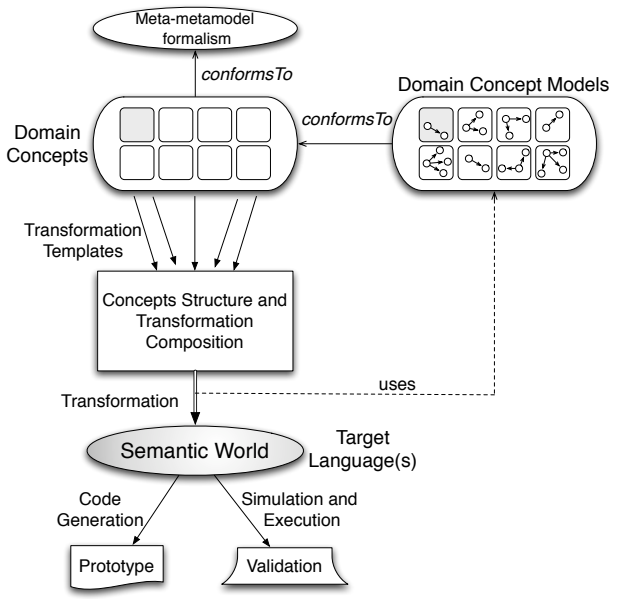


Figure 1. DSML Design Strategy High Level Overview

language that is precise and provides a well defined semantics. The *domain concept* can be seen as a brick that represents a basic idea that can be present in one or several DSMLs. A domain concept is an artifact used to express a concept and that can be applied to other domain concept (or even a pre-defined DSML) in order to extend it.

This definition of domain concept is very expressive in the sense that rather than only defining an abstract syntax and associating a structural semantics, we work at a level that semantics is provided (by transformation) for each domain concept.

2. LANGUAGE DESIGN STRATEGY

Metamodels represent an abstraction in the way of understanding a particular scientific or engineering domain. They are also known as the abstract syntax of a language. They provide the embodiment of modeling paradigms such as notions, ideas, abstractions, structural constructs, behavioral axioms and constraints. The DSML development life cycle being described is firstly based in a library of domain concepts that provide abstractions for supporting design of some types of DSMLs and, in a later stage, on the composition and parameterization of them.

The Figure 1 provides an high level overview of the methodology presented in this article.

On the top of the Figure is represented the meta-metamodeling formalism: the root for defining metamodels. The box *Domain Concepts* represents the metamodels of each domain concept. These metamodels are instances of the meta-metamodeling formalism and have an associated transformation template $Tr_{dc}(m)$ here represented by the vertical arrows coming out from the box. Transformations are the operations that allow, by construction, to provide the necessary semantics to the domain concepts. The transformation of each one of the blocks is a self contained and atomic operation.

For being able to define a DSML the metamodels from the pre-defined library must be composed. This step is done by either using compositional operators or by means of parameterization of the domain concepts. The *Concepts Structure and Transformation Composition* rectangle represents the act of mixing the concepts according to the chosen strategy - Section 3 details this process.

The list of pre-defined domain concepts provided so far can be found in (Software Modeling Verification 2008) and they include concepts such as:

- Data Structural Patterns
  - Data Type
  - Data Structure
- Control Structure Patterns
  - Assignment
  - Conditional Statement
  - Iteration
- Behavioral Patterns
  - Finite State Machine

It should be noted that the methodology under description can be applied to other concepts and in particular to existing DSMLs (that must be extended in some sense). The limitation of the approach is technical and forces that transformations defined into the target language to respect a template of transformation. This is due to the fact that transformation process will interact with each other at some point and this process must be controlled. Each transformation is defined as a set of other transformations $Tr_{mm} = \{Tr^1_{mm}, Tr^2_{mm}, \ldots, Tr^n_{mm}\}$ each one of them corresponding to the rule(s) of transforming certain elements of the source model into elements of the target model. The mm index defines to what source metamodel transformation is related to.

The domain models (that maintain a conformity relationship between the domain concepts) are also taken as an input of the final transformation process represented by the double lined arrow in Figure 1.

Whenever target language(s) provide prototyping and verification capabilities, the transformed DSML will be used for prototype generation and verification and, some times, for automatic test case generation.

3. PARAMETERIZATION AND COMPOSITION

The idea of parameterizing and composing metamodels and transformations relies on a language driven engineering approach. Each DSML is built by using an incremental and modular approach.

The main goals behind the idea of composing metamodels is to (Jackson and Sztipanovits 2006):

- Manage language complexity;
- Re-use of concepts for faster language development;

More precisely, to re-use the concepts that have been defined in order to create more complex language structures based on them. Some of these concepts are addressed in (Object Management Group members 2007), (Ledeczi et al. 2001) or (Emerson and Sztipanovits 2006) but usually only in a syntactic point of view.

Parameterizing acts as replacing an existing element in the metamodel by another compatible one, whereas composition is the act of "gluing" together different metamodels. For both of the approaches, our proposal implies that the transformations must also be adapted.

3.1 Parameterization

Parameterization is the action of enriching a metamodel by means of another metamodel. A parameterization is defined as follows.

DEFINITION 1. *Metamodel parameterization: At the metamodel level a parameterization is defined as,*

$$mm' = mm[fp \xleftarrow{\varphi} ep, F_{fp}]$$

where

- mm, mm', fp, ep *are metamodels;*
- $ep \supset \varphi(fp)$ *re-defines, at least, the elements in* ep
- φ *is a total function that creates a map between elements of* fp *and* ep

  $$\varphi : fp \rightarrow ep$$

  in order to establish the replacement of nodes (classes) and references (associations, aggregations and generalizations);
- F_{fp} *is a set of formulas representing constraints over* fp *that must be respected..*

The fp metamodel defines a template of what can be replaced in the metamodel mm. It is obviously a subset of the metamodel mm which, in the case of the metamodeling formalism used for this article, is a set of ECore (Eclipse 2007) classes (UML Class Diagram like metamodelling language) and their relations.

The formal parameter fp is then replaced by an effective parameter ep.

A simplified diagram of the metamodel parameterization is presented in Figure2, which shows that a DSML metamodel is extended by defining its formal parameter and by substituting it with an effective parameter.

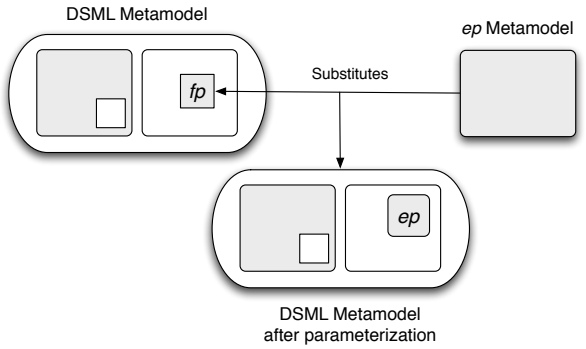


Figure 2. Metamodel extension by parameterization

As an example of a parameterization of metamodels lets consider the `Finite State Machine` and `Data Structure` domain concepts.

Let's consider that a DSML must have a kind of finite state machine that can be captured by the metamodel in the Figure 4. This metamodel does not specify anything concerning the `StateType`'s exact meaning. In order to provide this element with a richer semantics it is possible to extend it with the concept of `Data Structure` defined in Figure 3.

In terms of a parameterization this can be expressed by defining:

- mm the `FSM` metamodel;
- fp the metamodel subset of mm that, in this case, is the `StateType` element - the element to be parameterized;
- ep being the `Data Structure` metamodel in Figure 3.
- $\varphi = \{\langle StateType, DCDataStructure \rangle\}$

The effective parameter ep is, in fact, a metamodel that defines more elements that the ones defined by fp. This is what makes the

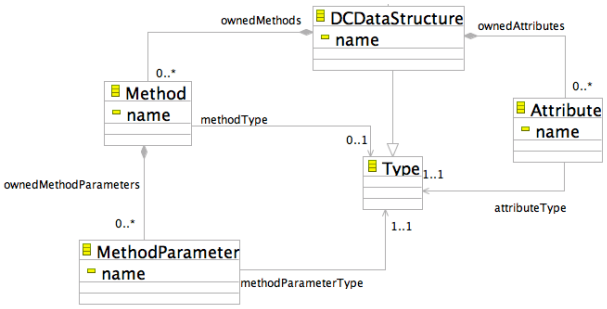


Figure 3. `Data Structure` Domain Concept

difference between fp and ep: fp defines the minimum template and ep provides the real metamodel to serve as concrete parameter used for instantiation. In this case no conditions are provided.

With this parameterization and applying the transformation it is possible to simulate a final state machine in which their states can have an associated data structure. The result of applying parameterization of a `Data Structure` over a `Finite State Machine` is presented in Figure 5.

The definition and composition, in what regards to transformations, are presented in Section 4.2.

3.2 Composition

Besides parameterization of metamodels the methodology under development defines a set of composition operators. These operators work at a more syntactic level than the parameterization. Nevertheless, depending on the operator used, transformations are adapted to cope with operator's semantics.

Figure 6 shows a schematic overview of transformation composition of domain concepts. In addition to the transformation template defined for each domain concept, the transformation for the target language also uses the definition of each operator and the semantics defined for each one of them.

Taking into account that the Λ is the set of operators available for composing domain concepts, a composition is generically defined by:

- mm_l is a metamodel domain concept acting as left-side metamodel;
- mm_r is a metamodel domain concept acting as right-side metamodel;

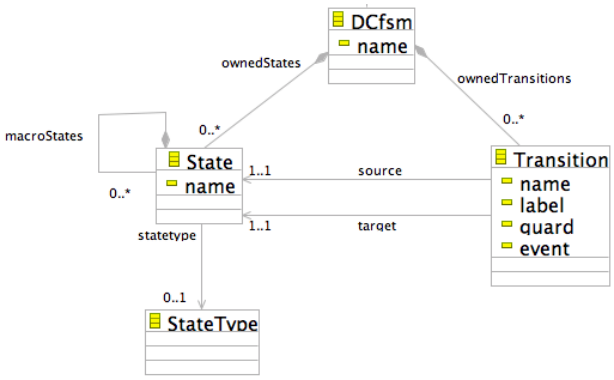


Figure 4. `FSM` Domain Concept

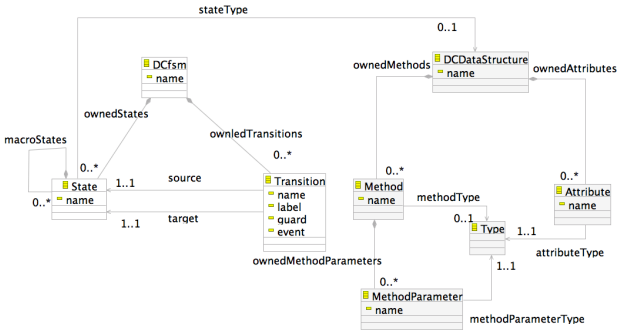


Figure 5. Metamodel Parameterization Result between `FSM` and `Data Structure`

- $op \in \Lambda$ the operator used in the composition;

A composition is thus defined as

$$mm' = mm_l \; op \; mm_r$$

The composition operators available are listed as follows:

Union: This operator takes two metamodels and produces a new one with the contents of both. In the case of elements with the same name duplication is performed: $mm' = mm_l \cup mm_r$

Merge: Same as `Union` with the specificity that classes, attributes and other constructions in the metamodel with same name are merged together. This operator can be defined as: $mm_l \backslash (mm_l \cap mm_r) \cup mm_r \backslash (mm_l \cap mm_r) \cup (mm_l \cap mm_r)$.

Using these last two operators implies to create a merge of the left and right-side specifications in the target language.

Inherit: The inheritance operator allows to compose metamodels with *UML-like* inheritance concept. The parameters for this operator are:

`specializedClass` as ec_l and `specializationClass` as ec_r: allows to specify if the class that is specialized (in the left metamodel) and the class that performs the specialization (in the right metamodel);

This operator creates a specialization relationship between ec_l and ec_r in the target formalism.

Implementation Inheritance: With this operator the children inherits all of the parents attributes, but only the containment associations where the parent acts as the container. No other associations are inherited (Ledeczi et al. 2001).

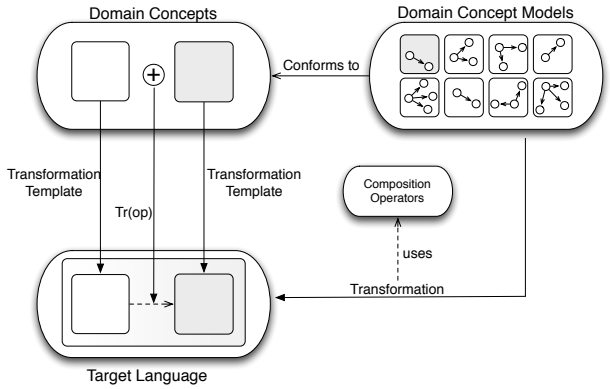


Figure 6. Transformation Composition

Interface Inheritance: This operator means that the inheritance allows no attribute inheritance, but does allow full association inheritance, with one exception: containment relations where the parent functions as the container are not inherited (Ledeczi et al. 2001).

Associate: This operator creates an association relationship between a class of one metamodel and another. This operator is parameterized as follows:

a) ec_l and ec_r for the name of the classes that are going to be associated;

b) `leftCardinality{0..*, 1, 1..*}` and `rightCardinality{0..*, 1, 1..*}` for defining the cardinality of the relationship.

The resulting transformation of applying this operator creates another Class that manages references of left and right sides accordingly to the multiplicity. This Class can be seen as a generic way of defining $n - n$ relations with $n \in \mathbb{N}_0^+$;

Containment This operator allows to create containment relations in order to provide hierarchical constructs. The hierarchical relation is given from one construction of the left metamodel to a set of constructs in the right metamodel. The parameters for this operator are the following:

a) ec_l as the name of the class in the left metamodel that acts as container;

b) $\langle ec_r, cardinality \rangle$ being a list of pairs class name (in the right metamodel) and cardinality of the containment relation.

4. TRANSFORMATIONS AS COMPOSITE SEMANTICAL BLOCKS

For a given DSML or domain concept there are associated transformations that provides the necessary semantics.

DEFINITION 2. *Transformation is a function* $Tr : im, ctx \rightarrow im', ctx'$, *where* im *is a model in the source DSML,* im' *a model in the target formalism,* ctx *and* ctx' *the system's execution state before and after transformation.*

$\Pi : t \rightarrow ctx$ *a function that returns the context of the system after applying transformation* t.

DEFINITION 3. *Set of Transformations:*

Having $im \in \mathbf{IM}|_{\mathbf{mm}}$ *where* $\mathbf{IM}|_{\mathbf{mm}}$ *is the universe of models that are in conformity with the metamodel* $mm \in \mathbf{MM} the universe of metamod$
A transformation is defined as a set of other transformations:

$$Tr_{mm} = \{Tr^1_{mm}, Tr^2_{mm}, \ldots, Tr^n_{mm}\}$$

The application of the of transformation to a given $im \in \mathbf{IM}|_{\mathbf{mm}}$ is:

DEFINITION 4. *Having a transformation* $Tr_{mm}(im)$ *a set of transformations* $t(im) \cup T_{mm}(im)$ *it is the equivalent to apply:*

$$(t \cup T)(im) = t(im, \emptyset) \cup T_{mm}(im, \Pi(t))$$

4.1 DSML Definition

Having formalized a transformation let us continue by stating the definition of a DSML and a parameterized DSML.

DEFINITION 5. *A DSML is as a 3-tupple*

$$DSML = \langle mm, F, Tr_{mm} \rangle$$

DEFINITION 6. *A parameterized DSML as a 4-tupple:*

$$DSML_p = \langle mm, fp, F_{fp}, Tr_{mm} \rangle$$

where F_{fp} *is the set of constraints over* fp.

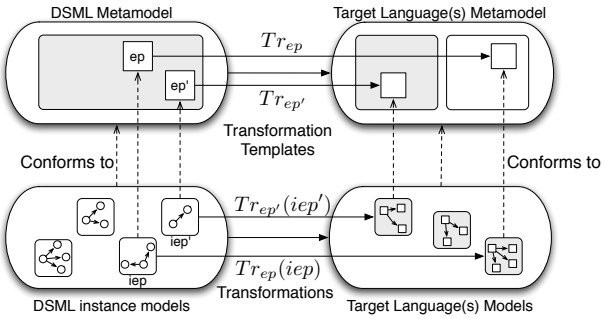


Figure 7. Transformation Parameterization

4.2 Integrating Transformations

We have focused so far in this article on the domain concepts and on the fact that a transformation template exists for each one of them, describing how to compose their allowing to produce a resulting metamodels. However, this preliminary result is still tackling the problem at a syntactic level: the result lacks of semantic information.

The next two steps are to produce models that are conform to the newly generated metamodel and to be able to re-use the pre-defined transformations for each one of the domain concepts. To accomplish this last objective two conditions must be fulfilled:

1. each transformation must be able to recognize parameterizations or compositions defined over domain concepts. In other words, each transformation has a structured attribute containing the information of the parameterization or composition defined;
2. a higher level transformation manager handles the execution of the transformation according to the parameterization and composition defined;

4.2.1 Parameterized Transformations

In order to be able to compose the DSML semantics the transformations are also composed according to the defined parameterization.

DEFINITION 7. *A parameterized transformation is:*

$$\forall im \in \mathbf{IM}|_{\mathbf{mm}}, \forall im' \in \mathbf{IM}|_{\mathbf{mm'}}$$

$$Tr_{mm'}[Tr_{fp} \xleftarrow{\varphi,\psi} Tr_{ep}] : im, ctx \rightarrow im', ctx'$$

where:

- mm' *is the metamodel result from the parameterization;*
- Tr_{fp} *the template transformation defined for the metamodel* fp*;*
- Tr_{ep} *the template transformation defined for the* ep*;*
- φ *the source mapping function:* $Dom(Tr_{fp}) \rightarrow Dom(Tr_{ep})$*;*
- ψ *the target mapping function :* $Cod(Tr_{fp}) \rightarrow Cod(Tr_{ep})$*;*

This means that the formal parameter fp relates with ep by φ in the source DSML domain. The transformations $Tr_{fp}(ifp)$ and $Tr_{ep}(iep)$ are related with each other by ψ in the target language domain.

By following these rules it is possible to obtain models in the target language without having to modify existing transformations or to re-define new ones providing a high level of re-usability.

The Figure 7 shows how parameterization works in a graphical representation of it. This ilustration details one of two processes present in white ellipse labeled *Concepts Structure and Transformation Composition* in Figure 1.

As depicted from this figure each domain concepts has a transformation template associated. In conjunction with the models that conform to the domain concept it allows to have specifications in the target language. Each domain concept might be parameterized by another domain concept or by a structure that also has a transformation defined. This transformation is represented by the arrow that goes from the smaller box on the top to the smaller box on the left bottom side.

In other words, if the formal parameter fp relates to ep by φ in the source DSML domain, and if we apply the transformations Tr_{fp} and Tr_{ep} to instances ifp and iep of their respective metamodels, then ψ expresses the relation between $Tr_{fp}(ifp)$ and $Tr_{ep}(iep)$ in the target language domain. From the operational point of view, ψ defines what transformations in fp are replaced by what transformation in ep.

Figure 7 resumes how the mm, fp, ep and mm' metamodels and their transformations relate to each other. The arrows marked with π represent projection of metamodels: the mm' metamodel, for example, if projected by mm (i.e. πmm), gives the grey part of mm, i.e. the part that does not include fp.

4.2.2 Transformations in Composition

The composition of transformations when a DSML metamodel is defined by means of composition operators is a more syntactic operation.

DEFINITION 8. *A transformation using compositional operators is:*

$$Tr(Tr_{ml} op Tr_{mr}) = Tr(op)(Tr_{ml}, Tr_{mr})$$

where $Tr(op)$ *represents the transformation template for an operator* $op \in \Lambda$.

In what concerns composition of domain concepts by using one of the pre-defined operators the transformations are treated as follows:

5. MODELS IN THE TARGET FORMALISM

For the target language we chose Concurrent Object Oriented Petri-Nets (CO-OPN) (Buchs and Guelfi 1991, 2000; Biberstein 1997). This formalism has been chosen because it is a formal language that allows to generate executable specifications and its Integrated Development Environment (IDE) provides a set of tools for simulation, verification and test generation (Lucio et al. 2006).

CO-OPN can be considered a General Purpose Language (GPL) encompassing very abstract and generic concepts. It is an object-oriented formal specification language based on synchronized algebraic Petri nets. Originally designed to support the specification of large distributed systems, it allows the definition of active concurrent objects and includes facilities for sub-typing, sub-classing, and genericity. There are various reasons why we argue that COOPN is suitable to be chosen as an intermediate format. Some of the more relevant are:

- It is a modular specification language allowing to specify different DSML components and their relationships;
- The specifications are described in a completely abstract axiomatized fashion;
- The system states can be completely defined and explored;

Basically, CO-OPN has three types of modules: *ADT(algebraic Abstract Data Types)*, *class*, and *context*:

- *ADT* represents data and its associated operations;
- *Class* is an encapsulation of algebraic Petri nets that allows to describe both structure and component's behavior. A CO-OPN class is generally composed by *methods*, *gates* (that can be seen as the return values for methods) and *places* that can by typed;

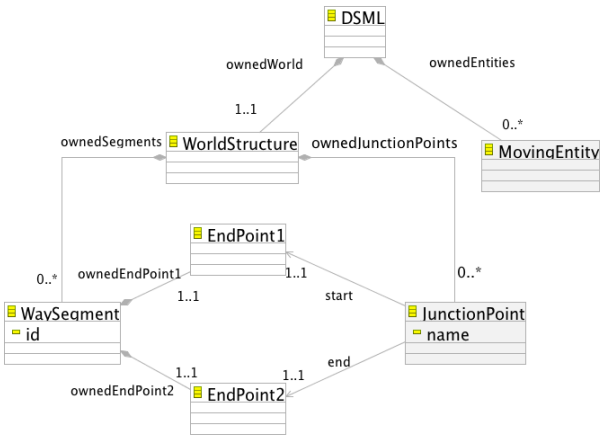


Figure 8. Generic Moving Entities DSML Metamodel $mmDSML_{gen}$

- *Context* is a higher level of encapsulation which defines the contextual coordination between components.

In addition, having CO-OPN has a target language for the implementation of this methodology allows to use several language features that, in the context of language prototyping, would had to be transformed into a more concrete language. These features include, for example, the notion of concurrency and transactions that are part of the CO-OPN language. They are transformed into an equivalent semantics in Java by COOPNBuilders' prototype generator.

6. EXAMPLE

Let us define a example for this paper in order to support our line of thought. The space limitation forces us to use a very simple system but that illustrates the methodology's application. Suppose we want to define a modeling language for the purpose of specifying the simulation of moving entities. Although we might understand the general principle that suits several domains demanding for DSMLs to implement their concepts (like train systems control, street traffic control, etc.), we do not have yet the full details to completely describe a DSML.

The general concepts involved, that might influence the syntax description of a DSML, could be described as having both **World Structure** information, and the mentioned moving **Entities** (i.e. trains, cars, etc).

The **World Structure** could be composed by **Junction Points** and **Way Segments** (i.e. cross-roads, etc) that are responsible to connect **Way Segments** (that depending on the domain could be particularized to rails, streets, channels, etc). Each **Way Segment** is composed by two end points, each of them could be connected to one **Junction Point**. However how many Segments are plugged to junction points, we define as rule that we can not plug both end points of a given to the same **Junction Point**.

The corresponding metamodel of the general concepts described previously are depicted in Figure 8.

In the next subsections we will have examples of Specific Languages for particular domains of Train Systems and Robot Systems.

Lets define $mmDSML_{gen}$ as the metamodel in Figure 8. This metamodel has an associated transformation:

$$Tr_{mmDSML_{gen}} = Tr_{WorldStructure} \cup Tr_{MovingEntity}$$

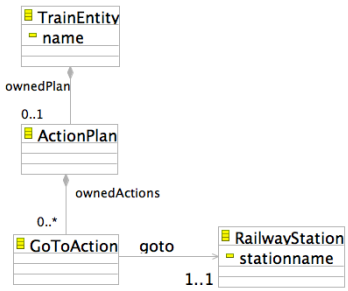


Figure 9. The Train Entity Metamodel $mmTrain$

and

$$Tr_{WorldStructure} = Tr_{WaySegment} \cup Tr_{Junction}$$

These transformations characterize the sequence of transformations performed to the CO-OPN language:

$Tr_{WorldStructure}$ creates a CO-OPN context representing the interface and the world in which the segments, moving entities and junction points are managed;

$Tr_{WaySegment}$ generates a CO-OPN Class with CO-OPN places representing the end points. An object of this class is also generated in the `WorldStructure` context;

$Tr_{Junction}$ a CO-OPN class that allows to map EndPoint1 to EndPoint2 and an object of this class type in the `WorldStructure` context;

$Tr_{MovingEntity}$ implies the creation of a another CO-OPN class for each one of the moving entities in the model.

By applying the definition of parameterized DSML:

$$DSML_{gen} = \langle mmDSMLgen, fp, F_{fp}, Tr_{mmDSML_{gen}} \rangle$$

6.1 A DSML for describing a Train System

Now that we want to define a modeling language for the purpose of specifying the simulation of a very simplified railway system. The particularization of our previously defined Entity in the general metamodel to the concept of train is depicted in Figure 9.

Basically, we define the concept **Train** as having a Structure with an attribute Name, and the behavior as an **Action Plan**.

A possible particularization of the concept **Junction Point** could be to the concept of Railway Station.

The referred **Action Plan** is a sequence of possible GoTo actions. Informally the behavior of a GoTo action is to send the a particular train to a given Railway Station.

Having $mmTrain_{train}$ the metamodel corresponding to the Train System, we define it as follows:

- fp the metamodel corresponding to the `MovingEntity` and `JunctionPoint` elements of $mmDSML_{gen}$;
- $mmTrain$ effective parameter as the metamodel in Figure 9;
- $\varphi = \{\langle MovingEntity, TrainEntity \rangle, \langle JunctionPoint, RailwayStation \rangle\}$

The result of the metamodel parameterization is presented in Figure 10 with the new and affected elements prior to transformation with a grey background. As it was previously introduced, the new metamodel is obtained by applying:

$$mmDSML_{train} = mmDSML_{gen}[fp \xleftarrow{\varphi} mmTrain, F_{fp}]$$

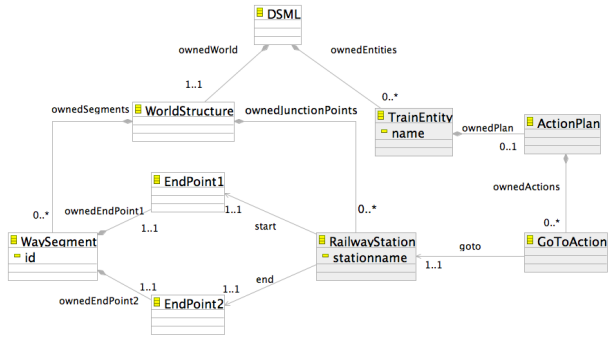


Figure 10. Railway System Metamodel

F_{fp} constraints are not applied in this example.

Followed to the composition of the DSML structure, existing models for both $mmDSML_{gen}$ and $mmTrain$ are regenerated in conformity with the new metamodel. The next step is to transform the $mmDSML_{train}$ by re-using the existing transformations.

Taking into account that $Tr_{mmDSML_{train}}$ is defined as a sequence of transformations:

$$Tr_{mmTrain} = Tr_{TrainEntity} \cup Tr_{RailWayStation}$$

where:

$Tr_{RailWayStation}$ a CO-OPN class that allows to map EndPoint1 to EndPoint2 with another CO-OPN place acting as the `name` attribute. An object of this class type is also generated in the `WorldStructure` context;

$Tr_{TrainEntity}$ creates in the target language a CO-OPN class with a place for its name, another one for storing the information corresponding to the train status (e.g. in which `RailwayStation` it is, and a method to implement the `GoToAction` behavior.

by using the definition in Section 4.2.1:

$$Tr_{mmDSML_{train}} = [Tr_{fp} \xleftarrow{\varphi,\psi} Tr_{mmTrain}]$$

meaning the the final sequence of transformations is, by applying φ:

$$Tr_{DSML_{train}} = Tr_{WaySegment} \cup Tr_{RailwayStation} \cup Tr_{TrainEntity}$$

The train DSML is thus given by:

$$DSML_{train} = \langle mmDSML_{train}, F, Tr_{DSML_{train}} \rangle$$

The application of the transformation to a $im \in \mathbf{IM}|_{\mathbf{mmDSML_{train}}}$ is:

$$im' = Tr_{WaySegment}(im, \emptyset) \cup Tr_{RailwayStation}(im, \Pi(Tr_{WaySegment})) \cup Tr_{TrainEntity}(im, \Pi(Tr_{RailwayStation}))$$

6.2 A DSML for describing a Robot System

Suppose now that we want a DSML to describe the domain of robot systems. Let us define a very simplified robot system. The Robot has no particular way segment to follow, nevertheless the Junction Points can be seen as intermediate Pickable Object.

The sequence of possible actions for our Robot as defined in Figure 11 is: Start, Stop and Pick Object. The informal semantics associated to these three actions can be described in natural language in the following way:

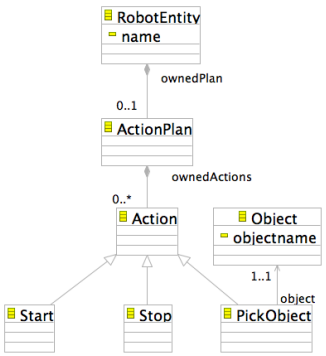


Figure 11. The Robot Entity Metamodel $Robot_{mm}$

- Start - to start moving the robot forward in order to reach the Pickable Object and make it disappear once reached. The robot stops immediatly after waiting for the next target. If no goal is set the robot does not move;
- Stop - to stop moving the robot;
- Pick Object - this action sets the target (Pickable Object) where the robot should move. In other words the robot gets the reference to the object to pick, rotating a certain angle in order to be facing the object and be able to mode forward in a straight line to find the object when the Start action is called.

Having $mmDSML_{robot}$ the DSML metamodel corresponding to the Robot System, we define it as follows:

- fp the metamodel corresponding to the `MovingEntity` and `JunctionPoint` elements of $mmDSML_{gen}$;
- ep the metamodel in Figure 11;
- $\varphi = \{\langle MovingEntity, RobotEntity \rangle, \langle JunctionPoint, Object \rangle\}$

The result of the metamodel parameterization is presented in Figure 12. The Robot System metamodel is obtained by applying:

$$mmDSML_{robot} = mmDSML_{gen}[fp \xleftarrow{\varphi} Robot_{mm}, F_{fp}]$$

In this case, $Tr_{mmRobot}$ is defined as a sequence of transformations:

$$Tr_{mmRobot} = Tr_{RobotEntity} \cup Tr_{Object}$$

where:

Tr_{Object} a CO-OPN class that allows to map EndPoint1 to EndPoint2 with a CO-OPN place acting as the `attribute` and another one holding the availability of the `object`. An object of this class type is also generated in the `WorldStructure` context;

$Tr_{RobotEntity}$ creates in the target language a CO-OPN class with a place for its name, another one for storing the information corresponding to the robot status (Running, Stopped), and three methods, each one for the different types of actions.

Applying the same definitions as for the Train System DSML, the transformation for $DSML_{robot}$ is give by:

$$Tr_{DSML_{robot}} = Tr_{WaySegment} \cup Tr_{Object} \cup Tr_{RobotEntity}$$

and the Robot DSML is thus given by:

$$DSML_{robot} = \langle mmDSML_{robot}, F, Tr_{DSML_{robot}} \rangle$$

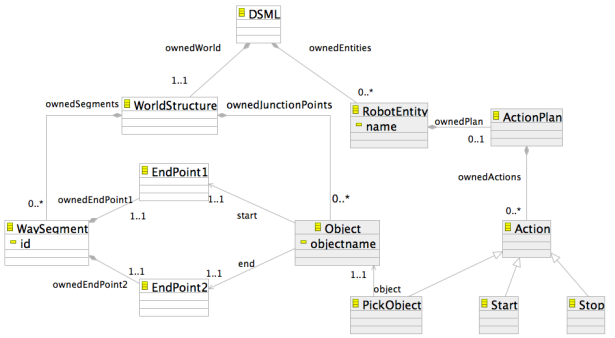


Figure 12. Robot System Metamodel

The application of the transformation to a $im \in \mathbf{IM}|_{\mathbf{mmDSML_{robot}}}$ is:

$$im' = Tr_{WaySegment}(im, \emptyset) \cup Tr_{Object}(im, \Pi(Tr_{WaySegment})) \cup Tr_{RobotEntity}(im, \Pi(Tr_{Object}))$$

6.3 Models in the Target Language

The generated specification in CO-OPN language allows to create a Java executable prototype that allows to simulate the behavior trains in their world. Depending on how much semantics is added to the transformation it is also possible to generate an executable specification that manages the creation of new trains/robots, new segments and railway stations/objects.

7. CONCLUSIONS AND FUTURE WORK

In the context of this article we presented a conceptual framework and methodology that allows creation of DSMLs for prototyping and verification. It provides syntactic and semantics composition of concepts allowing to define a specific DSML behavior by starting with a more abstract view of the language and then by particularizing some of its concepts to fit a more precise semantics. We are currently working in the integration of constraint definition and resolution in order to better control the substitution/parameterization mechanism.

This methodology goes in the line of the *extentionOf* concept presented in (Barbero et al. 2007) but not only considering the syntactical part of the language but also taking into account its semantical aspects. This allows to have a framework that is suitable for testing and verification purposes and that allows re-use of semantical components.

Considering the model extension by package merge (Object Management Group members 2005) in UML2 specification defined to modularize the UML2 metamodel, we choose to use a different approach mainly because this technique is very UML dependent and lacks of a precise definition.

The conceptual framework presented here is currently under development by using Ecore as source formalism, CO-OPN as the target and by implementing transformation management and composition with ATL (ATLAS Group 2008). We also expect to extend this methodology in order to support graphical aspects of the DSMLs using, whenever possible, the same compositional and parameterization approach.

References

ATLAS Group. Atlas transformation language, 2008. `http://www.eclipse.org/m2m/atl/`.

Mikaël Barbero, Frédéric Jouault, Jeff Gray, and Jean Bézivin. A practical approach to model extension. In David H. Akehurst, Régis Vogel, and Richard F. Paige, editors, *ECMDA-FA*, volume 4530 of *Lecture Notes in Computer Science*, pages 32–42. Springer, 2007. ISBN 978-3-540-72900-6. URL `http://dblp.uni-trier.de/db/conf/ecmdafa/ecmdafa2007.html#BarberoJGB07`.

Olivier Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems*. PhD thesis, University of Geneva, 1997.

Frederick P. Brooks. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

Didier Buchs and Nicolas Guelfi. A formal specification framework for object-oriented distributed systems. *IEEE Transactions on Software Engineering*, 26(7):635–652, july 2000.

Didier Buchs and Nicolasi Guelfi. A concurrent object oriented petri nets approach for system specification. In *12th International Conference on Application and Theory of Petri Nets*, pages 432–454, 1991.

Eclipse. Eclipse modeling framework, 2007. `http://www.eclipse.org/modeling/emf/?project=emf`.

Matthew Emerson and Janos Sztipanovits. Techniques for metamodel composition. In *OOPSLA - 6th Workshop on Domain Specific Modeling*, pages 123–139, Portland, Oregon, October 2006. ACM Press. URL `http://chess.eecs.berkeley.edu/pubs/289.html`.

Ethan Jackson and Janos Sztipanovits. Towards a formal foundation for domain specific modeling languages. In Wang Yi Sang Lyul Min, editor, *Proceedings of the Sixth ACM International Conference on Embedded Software (EMSOFT 06)*, pages 53–63. ACM, October 2006. URL `http://chess.eecs.berkeley.edu/pubs/286.html`.

D. A. Ladd and J. C. Ramming. Two application languages in software production. In *Proc. of the 19994 USENIX Symposium on Very High Level Languages(VHLL)*, pages 169–177, Santa Fe, NM, 1994.

Akos Ledeczi, Greg Nordstrom Gabor Karsai, Peter Volgyesi, and Miklos Maroti. On metamodel composition. In *Control Applications, 2001. (CCA '01). Proceedings of the 2001 IEEE International Conference on*, pages 756–760, Mexico City, Mexico, September 2001. IEEE Computer Society. doi: 10.1109/CCA.2001.973959. URL `http://dx.doi.org/10.1109\%2FCCA.2001.973959`.

Levi Lucio, Luis Pedro, and Didier Buchs. Semi-automatic test case generation from co-opn specifications. In *Proceedings of the Workshop on Model-Based Testing and Object-Oriented Systems*, pages 19–26. Microsoft Research, 2006.

Object Management Group members. Meta-Object Facility 2.0 core specification. Technical report, OMG, January 2007. URL `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`. `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`.

Object Management Group members. Uml 2.0 superstructure specification. Technical report, OMG, August 2005. http://www.omg.org/cgi-bin/doc?formal/05-07-04.

Group Software Modeling Verification. Library of domain concepts, 2008. `http://smv.unige.ch/tiki-index.php?page=MTVLibraryDomainConcepts`.

Vanderbilt University, Institute for Software Integrated Systems. *GME 5 User's Manual*. Vanderbilt University, 2005.

A Domain-Specific Approach to the Development of Ontology-Based Document Assessment Assistants

Arturo J. Sánchez-Ruíz
School of Computing – University of North Florida
1 UNF Drive, Jacksonville, FL 32246, USA
+1-904-620-1314
asanchez@unf.edu

Bart Welling
English Department – University of North Florida
1 UNF Drive, Jacksonville, FL 32246, USA
+1-904-620-1268
bhwellin@unf.edu

ABSTRACT
The second author of this paper incrementally developed, over the years, a manual process to systematically evaluate English essays by maintaining an ontology of comments crafted with the goal of pinpointing the occurrence of mined negative writing patterns (i.e., those whose use is discouraged), as well as positive ones (i.e., those whose use is praised). In this paper we report on how the automation of this process, using domain-specific approaches, led to the development of an ontology-based assessment assistant for a specific word processing system. We also report on our approach, currently underway, to extending this solution to a product line of ontology-based assessment assistants over a family of word processing systems and course management systems.

Categories and Subject Descriptors
D.2.6 [**Software Engineering**]: Programming Environments – *integrated environments, interactive environments.* D.2.11 [**Software Engineering**]: Software Architectures – *domain-specific architectures, patterns.* D.2.13 [**Software Engineering**]: Reusable Software – *domain engineering, patterns.*

General Terms
Management, Design, Human Factors.

Keywords
domain-specific software development; document assessment; software assistants; ontology-based software; software product lines.

1. INTRODUCTION
The inception of this project can be traced back to an informal conversation between the two authors in the summer of 2006, during which the second author related an assessment method he had developed over the years after having evaluated hundreds of English essays. In Section 2, the second author explains how he synthesized his method.

The first author saw the possibility of completely automating this method by constructing a software assistant which would enable evaluators to systematically, consistently, and efficiently assess documents from various domains, not just English essays, e.g., legal, governmental, medical, technical, et cetera. For instance, the assistant would enable evaluators to maintain and refine their knowledge of observed writing patterns used by authors on their documents, positive and negative alike, a practice which promotes uniformity in assessment, and the sharing of such knowledge.

This paper reports on three evolution stages associated with this project. In its first stage the project was a capstone-like course assignment. In its second stage, the project was a product developed for a specific word processing system. Finally, in its third stage, the project—currently underway—is to develop a product line of ontology-based assessment assistants, over a family of word processing products and course management systems typically used in educational institutions, which give end-users the feeling of working with a unified tool which assists them in the management, delivery, and assessment of documents with minimal context switching.

Section 3 shows the domain analysis associated with the first stage of this project which allowed us to identify the major components of the system and suggested strategies to incrementally develop the project.

Section 4 discusses the software architecture associated with the third stage of this project.

Section 5 discusses implementation details associated with the development of an assessment assistant which targets the Microsoft Word product.

Section 6 presents our approach to generalize this specific solution to a software product line over various word processing products and course management systems.

Section 7 compares our approach with others with respect to: the original goals of the assistant as envisioned by the second author; course management systems; and approaches to build software product lines.

The paper ends with our conclusions and with references to the literature we consulted.

2. User's Story
Shortly after being hired to teach literature at the university level, the second author realized that much of his time outside the classroom would be spent grading student essays. Unfortunately, too many of these essays exhibited exactly the same negative writing patterns: weak and poorly structured arguments; ineffective handling of primary evidence; incorrect citation of outside sources; unclear, clichéd, or overly simplistic sentence structures and word choices; and major, copious errors in spelling, grammar, and punctuation. Commenting on these problems in the traditional way, with pen in hand, proved to be not only labor-intensive and frustrating but pedagogically inefficient as well.

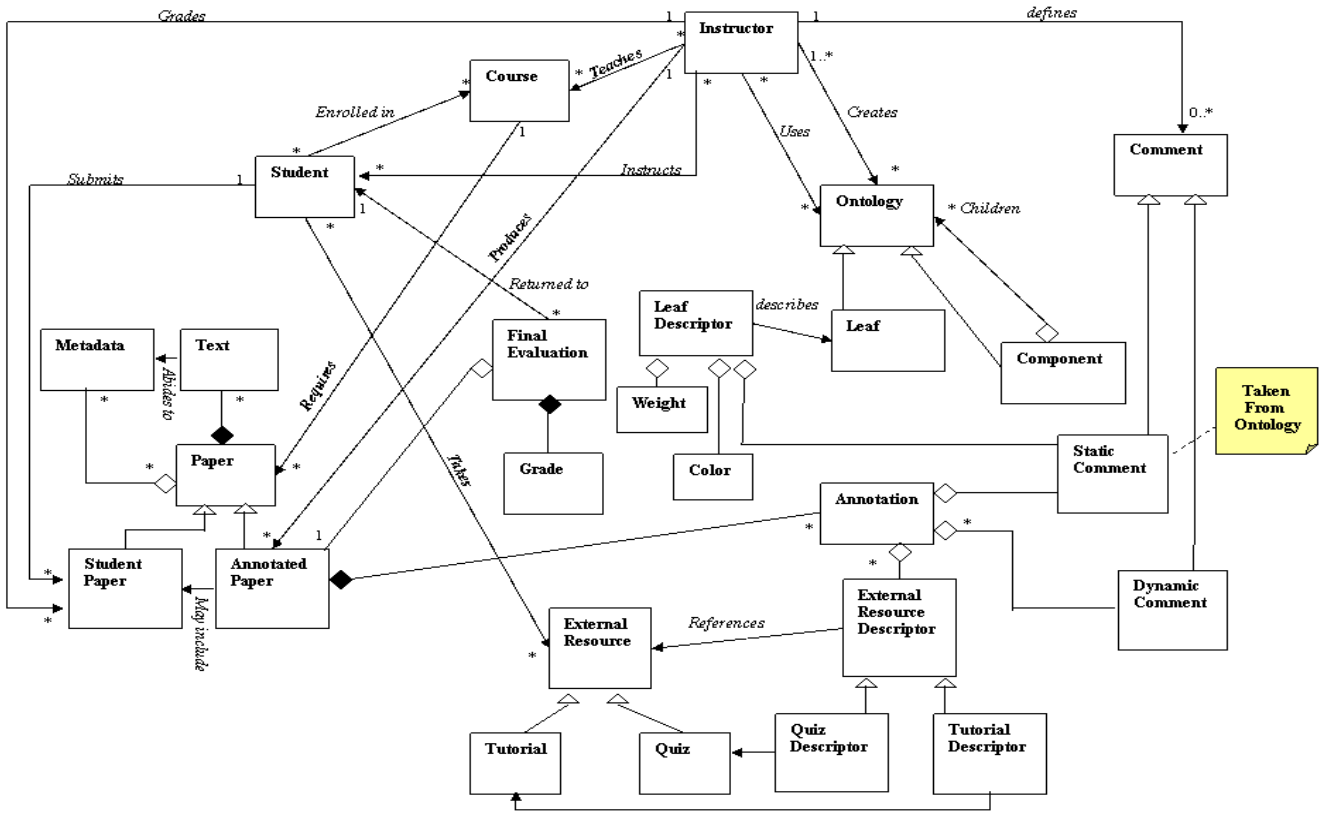


Figure 1. Domain model as an UML class diagram.

Students with the most serious writing challenges seemed to feel overwhelmed by the sheer number of comments, and given the space constraints of handwritten comments it was frequently possible to highlight only what was wrong with a given feature of a student's essay rather than how to fix it—or, better yet, how to avoid the problem in the future.

Students also complained about what they perceived as mixed messages from teachers in different academic departments, and sometimes within the same department, regarding the university's criteria for strong student writing. It quickly became apparent to the second author that students and faculty members alike lacked a common critical vocabulary for evaluating, and improving, the quality of written communication at the university. Moreover, they lacked a method to help formalize, standardize, streamline, and enrich the process of communicating with each other about student writing skills. Microsoft Word's Comment feature presented some advantages over handwritten comments, but in the end it still required the user to type the same comments in paper after paper, and time constraints made it difficult to add comments of more than a few words in length. More seriously, Word's Comment feature was not designed to link comments together within a pedagogically meaningful and effective framework of ideas regarding the nature of strong, clear writing.

The second author began developing a set of "Grading Codes"[1] that could be inserted directly in students' essays either by hand or via a Microsoft Word comment. The codes consist of short, simple combinations of letters and numbers which are keyed to detailed comments on virtually every aspect of student writing. For instance, the code "CL-1-b" corresponds to the comment "While at one point this sentence would have been considered debatable (i.e., a claim), it would now be accepted as true (i.e., a fact) by the majority of scholars, and should thus be rendered more debatable." The codes comprise an ontology that can be expanded every time the second author encounters a previously unknown error in his students' essays. Indeed, like the Linnaean taxonomic hierarchy, the "Grading Codes" have the potential to be infinitely expandable. However, the greatest strength of the system is also its greatest weakness. As the list of "Grading Codes" grows, it takes more and more time for students to decipher the comments on their papers, and the risk that they will simply ignore the codes increases.

After discussing the preliminary version of the "Grading Codes" with the first author, both authors realized that an automated assessment assistant combining the best features the "Grading Codes," and Microsoft Word's Comment feature would present the ideal solution to these problems. A software

[1] A sample of these codes can be downloaded from http://www.unf.edu/~asanchez/dsm08

application that interoperated with Microsoft Word would enable the user to add detailed pre-written comments to documents quickly and exactly where they were needed. Precise and transparent point values (determined and revised, as needed, by evaluators) could be assigned to different positive and negative writing patterns. The application, like the "Grading Codes," could be based on an infinitely expandable and flexible ontology, but students—while familiar with the ontology, and thus with the standards against which their writing was being evaluated—would not be distracted by the need to engage in context switching between a set of codes and the comments they stood for.

The assessment assistant could also incentivize student learning by means of hyperlinks to online tutorials and quizzes, the successful completion of which could translate into a higher grade on the essay. Not only would such an application foster clearer communication between teachers and students vis-à-vis student writing, but it could benefit academic departments and entire universities by facilitating greater uniformity in grading practices, by providing a mechanism for sharing customized grading ontologies, and by enabling academic units to track and respond more effectively to emerging trends in student writing. Finally, since users would be able to revise ontologies freely, the application could potentially streamline and enhance the document assessment process in any number of fields beyond the walls of academia.

3. DOMAIN ANALYSIS

The result of our initial domain analysis, using the approach discussed in the book by Larman [5], is presented in Figure 1 as an UML diagram [6]. Arrows with a small head represent general association relationships (labeled); arrows with a larger head represent "is-a" relationships (i.e., generalization/specialization); clear diamonds represent aggregation relationships; dark diamonds represent composition relationships; and boxes represent relevant domain concepts. Some multiplicities are shown.

This initial analysis allowed us to identify two major sub-systems: document management and delivery; and document assessment. The former interfaces with course management systems. The latter interfaces with word processing systems and maintains ontologies created by evaluators. In the next section we show these components depicted as an architectural diagram.

4. SOFTWARE ARCHITECTURE

Figure 2 presents a view of how the major subsystems identified by our domain analysis interact, from the perspective of end-users: document authors and document evaluators. The acronym ISA stands for Integrated Software Assistant.

We refer to the encircled numbers in Figure 2 to illustrate a typical use case. Authors prepare their documents using some word processing system, which we abbreviate as WPS (1). Authors submit their documents to the evaluators via the course management system, which we abbreviate as CMS (2). When evaluators launch the ISA on their computers, it connects to the CMS and determines whether documents are ready to be downloaded to their computers (e.g., if the current date is past the deadline associated with this coursework). In this case, documents are downloaded, typically as a single archive—e.g., zip, rar, flex, etc.—uncompressed and stored in a well-defined place from which ISA can retrieve them. At this point ISA is also keeping track of who submitted what, e.g., in connection with a coursework. The main user interface metaphor associated with ISA is that of a "dashboard" which contains icons suggesting various tasks evaluators can perform (3).

When evaluators are ready to start assessing the documents, they interact with ISA through the dashboard and additional dialog boxes, shown as needed. For a single document, the flow of activities is as follows. ISA opens up the document to be evaluated in the WPS. As evaluators identify the occurrence of a writing pattern, they select the text, and then the corresponding comment from the ontology. Evaluators can use multiple ontologies and update them in the middle of the evaluation process at will (4).

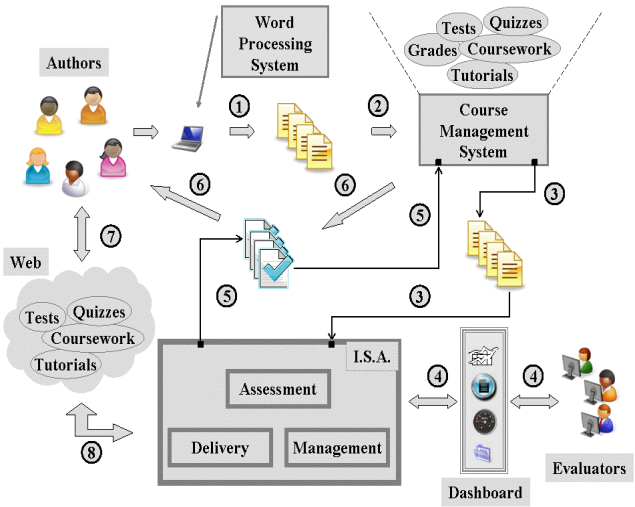


Figure 2. Software architecture: end-users' view.

When evaluators let ISA know they have finished evaluating all documents, ISA packs them as a whole, and sends them to the CMS. The final effect of this interoperation between ISA and the CMS is that documents are stored in pre-defined places in the CMS from which authors can retrieve them, and document grades (e.g., in the case of a course) are recorded as per the CMS's conventions, which become available for students to browse (5).

Authors connect to the CMS and retrieve their documents. To them the retrieved WPS document is just the original submitted document augmented with comments—inserted by ISA as per the evaluator's actions—and an extra page at the end with a summary of the evaluation, which includes the grade associated with this document, when applicable (6).

Since comments in the evaluated documents can contain embedded hyperlinks, ISA knows (for instance): if an author visited a website with a learning instrument (e.g., tutorial, quiz, and test); the author's identity; if the author completed the instrument; and a summary of the author's attempt (7). This is recorded by ISA as an assessment activity, which is transferred to

the CMS, and is therefore part of the author's assessment record, when applicable (8).

Various important design decisions were derived from the analysis of this software architecture:

[DD.1] The graphical user interface (GUI) exists in its own layer.

[DD.2] ISA interacts with the CMS via an interoperation layer.

[DD.3] ISA interacts with the WPS via an interoperation layer. This implies the solution should not be implemented as an add-in or plug-in to the WPS.

[DD.4] Ontology maintenance is independent of GUI, CMS, and WPS.

5. THE ONTOLOGY-BASED ASSESSMENT ASSISTANT FOR MS WORD

The first developed product is a specialization of the architecture in Figure 2 such that: WPS is Microsoft's Word (MSW), and ISA is just the Assessment component. The next two sections discuss implementation details associated with the Ontology Manager and the Interoperation Layer with MSW. The last section shows some screen shots of the product we developed.

5.1 Ontology Manager

Writing pattern ontologies are implemented as taxonomies. A taxonomy has a root which names the whole artifact. Under the root there are nodes which are either internal or terminal. Nodes are related by the category-subcategory relationship. Internal nodes must have descendants, which can be either internal or terminal nodes. Terminal nodes do not have descendants. The information associated with internal nodes is: the name of the category and references to descendants. The information associated with terminal node includes, but is not limited to: the name of the final category, the comment associated with the pattern, its weight, and URL's to external instructional resources.

From the perspective of the user, taxonomies can be directly manipulated via operations on categories which include: insert, delete, edit, and move. They can also be imported and exported. Taxonomies can be in two states: design, and publish. The first mode characterizes a work in progress. The second mode characterizes a finished product. Taxonomies can be transitioned from one mode to the other. They are presented to the user as a hierarchical structure with nodes that can be expanded, contracted, and moved.

Internally, taxonomies are represented as XML files with an associated schema. Imported taxonomies are checked against such a schema. Finally, schema well-formedness is the criterion used to determine whether taxonomies can be transitioned from design to publish mode.

5.2 Interoperation Layer with MS Word

Visual Studio Tools for Office (VSTO), currently in its 2005 version[2], enhances the popular MS Integrated Development Environment (IDE), Visual Studio[3], by enabling the seamless run-time interoperation between MS Office[4] applications and solutions built with the IDE.

The so-called Primary Interop Assemblies (PIA) act as the interoperation layer between our Assessment Assistant and the MS Word application. The PIA exposes the MS Word application itself and its run-time object model through .NET managed code. We decided (c.f. Design Decision [DD.3] on Section 4) to implement the Assessment Assistant as a separate application from MS Word, which interoperates with it via the PIA. The book by Carter and Lippert, as well as the book by Bruney discuss other viable programming models [1, 2]. The existence of the PIA for MS Word implied we did not need to implement the interoperation layer ourselves.

5.3 Screen Shots of our Assessment Assistant

We built a self-contained installer that checks whether the host computer has the correct versions of MS Word (2003 Professional or newer), .NET framework (version 1.1 and newer), and the PIA which correspond to these two components. The installer sets up specific folders where design/publish ontologies are kept.

Figure 3 shows the first interaction dialog the user sees when the application is launched. Figure 4 shows the options made available to the user after s/he has chosen "Manage Taxonomy".

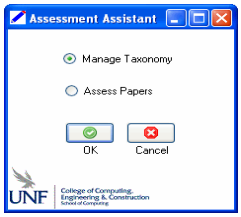


Figure 3. Initial options.

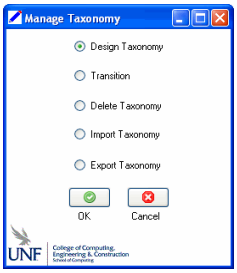


Figure 4. Manage Taxonomy options.

Figure 5 shows what the user sees after s/he has selected to continue working on an existing ontology in design mode, and has chosen the desired one. Notice the nodes can be expanded and contracted. Also notice the various options available.

[2] Visit the VSTO Developer Portal at http://msdn.microsoft.com/en-us/office/aa905533.aspx

[3] Visit the Visual Studio Developer Center at http://msdn.microsoft.com/en-us/vstudio/default.aspx

[4] Visit the MS Developer Center at http://msdn.microsoft.com/en-us/office/default.aspx?PHPSESSID=388057524368e3818e5a18783b5bd3fc

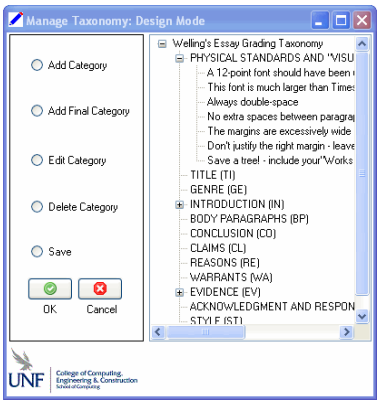


Figure 5. Working with an ontology in design mode.

When the user is ready to start assessing a document, s/he must first select the ontologies that will be used. After that, if the user chooses the "Assess Paper" option, then s/he can open as many documents as needed, and the ontologies dialog box remains open (see Figure 6, which shows two open ontologies).

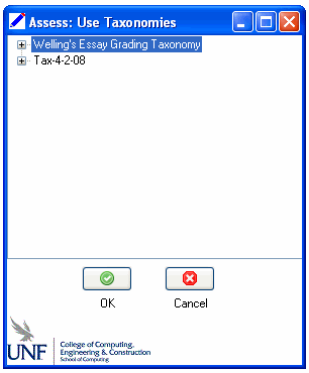


Figure 6. Ontologies to be used by the evaluator.

The user can switch back and forth among open documents, and close them at will. Only one document has the focus at any moment. Suppose the evaluator is now assessing the document with the focus, and s/he identifies a pattern. The user then selects the portion of the text with the mouse (left-click-hold-and-drag), goes to the ontologies dialog box to locate the appropriate pattern, clicks on it, and then the assistant automatically inserts a comment which contains all the information associated with the chosen terminal node in the ontology (see Figure 7).

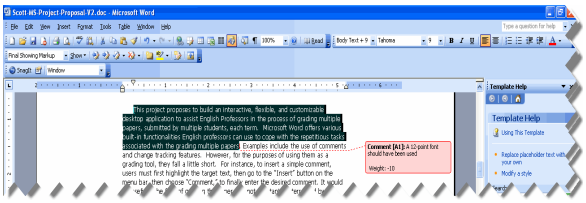
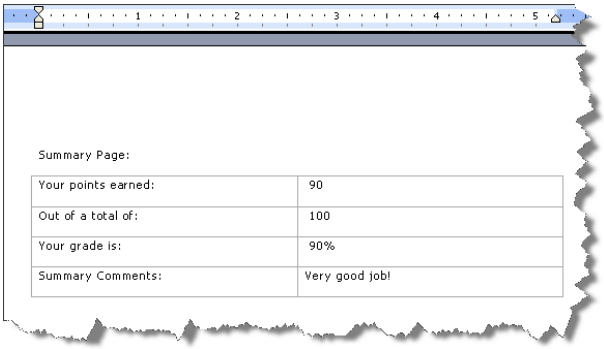


Figure 7. Pattern from the ontology inserted by the assistant.

When the user decides s/he has finished assessing the paper, the assistant generates a summary page and appends it to the document (See Figure 8). The user can revisit any evaluated paper at any moment if, for instance, s/he decides to change some of the comments and/or the associated weights.



Summary Page:

Your points earned:	90
Out of a total of:	100
Your grade is:	90%
Summary Comments:	Very good job!

Figure 8. Summary page generated by the assistant.

6. TOWARDS A SOFTWARE PRODUCT LINE OF ONTOLOGY-BASED ASSISTANTS

A software product line is defined by Clements and Northrop as "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [3].

From the perspective of the user, the common core of our product line is constituted, on the one hand, by a consistent user interface, hiding implementation details associated with desired features; and on the other hand, by the management of ontologies used in the assessment process. This common core is implemented, fundamentally, through a language of gestures—e.g., highlighting, pointing-and-clicking, clicking-and-dragging, et cetera—which allows users to directly manipulate ontologies and documents.

The variability of this product line can be projected onto two orthogonal axes, namely: that which characterizes the kind of word processor used by authors to compose their documents, and that which characterizes the course management system that frames the whole compose-assess-return document lifecycle.

We are therefore interested in assembling a family of ontology-based document assessment assistants as instances of the architecture discussed in Section 4. The targeted word processing systems (WPS) are OpenOffice and Acrobat. The targeted course management systems (CMS) are Blackboard[5], and Moodle[6].

To build the WPS interoperation layer (c.f. Section 4), we follow a reverse engineering approach. Namely, we first extract the calls to the Microsoft Office PIA (c.f. Section 5.2) and then build implementations of these calls against the Application

[5] Visit the Blackboard Developer Center at http://www.blackboard.com/extend/dev/

[6] Visit the Moodle portal at http://moodle.org/

Programming Interfaces (API) for OpenOffice[7] and Acrobat[8]. To build the CMS interoperation layer we follow a forward approach, namely we first define the appropriate interfaces and then implement them against the API's for the corresponding systems. This is because the operations associated with this layer are easy to understand as extensions of input/output services.

7. RELATED WORK

From the perspective of the Assessment Assistant's original goals as envisioned by the second author, we compared our approach with products from different educational technology companies to compile the information presented in a table (not shown here). We spoke directly with representatives from Vantage Learning (product: IntelliMetric "intelligent" Automated Essay Scoring System), Pearson Knowledge Technologies (product: Knowledge Analysis Technologies—KAT—Engine), and Idea Works (product: SAGrader). We also contacted Educational Testing Services (products: Criterion, E-Rater, Critique, and C-Rater), but they did not respond to our requests for more information.

The rubric used to compose the table is the following: (a) the tool automatically evaluates a document based on such criteria as strength of argument, structure, style, grammar, and/or spelling; (b) comments are added directly to the document; (c) evaluator's comments are distinguished from writer's text using a word processing system's comment tool (when available); (d) the tool fully interoperates with a word processing system; (e) the tool is potentially or currently applicable in a wide range of academic disciplines and business environments; (f) individual comments can be customized by the evaluator without outside help; (g) comments can be inserted quickly in documents; (h) comments are part of a larger ontology; (i) users can create a new ontology without outside help; (j) users can automatically import an ontology created by other users; (k) preexisting ontologies can be customized by the user; (l) an ontology can be exported to other users; (m) the tool employs an assessment metaphor that shows users how many texts have been evaluated, and in general it shows the progress of the evaluation process for a large set of documents; (n) the tool assigns point values to different patterns in the document; (o) the tool provides writers with an assessment page and/or grade; (p) the tool provides evaluators with a statistical analysis of assessed texts; (q) the tool implements its own unique Graphical User Interface (GUI); (r) the tool accommodates multiple languages; (s) the tool attempts to eliminate human-introduced errors, biases, and inconsistencies; and (t) the tool flags "problem essays," i.e., essays that cannot be scored by computer.

Our Assessment Assistant is fundamentally different from the programs produced by Vantage Learning, Pearson Knowledge Technologies, Idea Works, and Educational Testing Services in that it (1) does not attempt to evaluate the content of texts automatically; (2) uses word processing system's comments feature to directly add customizable comments to texts produced in said word processing system; (3) allows users to create, import, export, and customize ontologies without outside help; (4) employs an assessment metaphor that shows users how many texts have been evaluated; (5) does not attempt to eliminate human-introduced errors, biases, and inconsistencies; and (6) does not attempt to flag "problem essays," i.e., essays that cannot be scored by computer.

Notice that criteria (a), (s), and (t) require tools to deal with the semantics of submitted text, which our tool does not attempt to deal with. Our Assessment Assistant is a true "assistive tool" in the sense that it enables evaluators to perform their task faster in a consistent and systematic way. Our tool does not attempt to usurp evaluators and their expertise in the identification of used patterns and anti-patterns, an approach which has been the target of heated debates among members of the educational community.

From the perspective of features provided to end-users by course management systems, Blackboard (Bb) and Moodle do offer functionality that supports the management and delivery of document-centric coursework. However, the use of this functionality imposes an artificial context-switching on its users. It also potentially imposes the need to manually perform mechanical chores such as connecting to Bb/Moodle in order to download coursework, disconnect, do the preparation needed to start grading, grade, and finally connect again to upload the papers and also to upload the grades. With ISA, the user only needs to deal with one friendly and intuitive application for the assessment, management and delivery of the coursework in cooperation with the same tools students used to create and deliver it. The whole roundtrip, which starts with the submission of a paper and ends with a student receiving the graded paper and potentially taking advantage of additional learning instruments associated with the submitted paper, can be done with virtually no context-switching.

Bb also supports the development of assessment instruments such as tests and quizzes with the corresponding tracking of attempts made by students, by itself or by cooperating with other tools such as "Respondus"[9]. However, it currently does not support the tracking of attempts initiated from outside Bb. In this case ISA acts as a middleware between the users (teachers and students) and Bb as a provider of assessment instruments, therefore minimizing the context-switching for these users. This means users will continue using the features Bb provides and ISA will act as the mediator that helps users take advantage of these resources with minimal context-switching. In conclusion, the intention of this project is not to supplant existing WPS or CMS, but to build a software assistant that cooperates with them to offer end-users an extended, integrated, more powerful, friendlier, more efficient, and more effective tool.

We identify our approach to building a software product line of ontology-based assessment assistants, currently under development, with the "Product Parts" and "Assembly Lines" patterns documented in [3]. The first pattern is used in the building of core assets and the second is used when assembling core assets to produce an Assessment Assistant for specific variability choices. We also identify common points with the "use case development strategies" labeled as "forward evolutionary

[7] Visit the OpenOffice Developer's Wiki at http://wiki.services.openoffice.org/wiki/Main_Page#Getting_started_with_OOo_development

[8] Visit the Adobe Developer Connection at http://www.adobe.com/devnet/acrobat/

[9] See http://www.respondus.com/

engineering" and "reverse evolutionary engineering" by Gomaa [4], as described in Section 6.

8. CONCLUSIONS

This paper has presented an evolutionary approach to the building of a product line of ontology-based assessment software assistants. The first stage of the evolution used a domain-specific approach to identify the suite of concepts, their relationships, and operations with which end-users are familiar: documents, assessment ontologies, and an assessment process which directly manipulates documents and ontologies through gestures such as highlighting, pointing-and-clicking, clicking-and-dragging, et cetera. At this stage we designed a software architecture with clearly separated concerns: user interface, ontology management, document assessment, and document management.

The second stage of the evolution implemented an instance of this architecture by focusing on word-processing-system-neutral components, i.e., user interface and ontology management, and a word processing system interoperation layer specifically aimed at MS Word.

The third stage of the evolution, currently underway, implements the software architecture by taking into account the variability axes introduced by classes of word processing systems, and classes of course management system; via a combination of reverse and forward engineering. The target result is a family of ontology-based assessment assistants.

With respect to our application of domain-specific techniques to the development of this project, we would like to highlight the following. First, since our end-users—evaluators—directly manipulate objects naturally occurring in their domain of application—documents and ontologies—through a language of gestures, the elicited Domain-Specific Language (DSL) has therefore a non-textual syntax, which we did not formally define simply because we did not consider it a crucial contribution to the development of the project.

Second, this DSL is supported by two meta-models: the ontology meta-model (OMM) and the document meta-model (DMM). The OMM has been defined in Section 5.1 as a family of taxonomies with the category-subcategory relationship, and also illustrated as an instance of the Composite Design Pattern in Figure 1 (see the portion which contains "Ontology", "Leaf", and "Component"). Interestingly enough, the DMM is the MS Word Object Model [2].

Third—and final, the semantics of these models, i.e., what give meaning to a gesture such as "left-click-hold-and-drag" on a portion of a document—for instance, are given by the Primary Interop Assemblies in the case of DMM, and our implementation of the ontology manager in the case of OMM.

With respect to the level of generality of our approach, from the perspective of the domain of expertise associated with the documents to be assessed, our main argument is this: since the ontologies are created by domain experts—e.g., lawyers, accountants, physicians, et cetera—it is incumbent upon these experts, not upon our tool, to make sure the ontologies capture the appropriate writing patterns and anti-patterns. Experts are assisted by the tool, not substituted by it. This is why, in our opinion, the current state of the art of the area referred to as "Ontology Learning from Text"[10] is not applicable to the problem of automatically mining writing pattern ontologies from actual documents produced by authors. However, we do agree such an approach is worth exploring.

ACKNOWLEDGEMENTS

The development of this project was the subject of study in the sequence of graduate courses Engineering of Software I and II, offered by the School of Computing, University of North Florida, in the fall semester of 2006 and spring semester of 2007, respectively. We thank the students of this course for their contribution to the project (in alphabetical order): Lucas Downard, Swapna Mekala, David Scott, Sweta Shah, and Smitha Thomas. David Scott implemented the current version of the Assessment Assistant for Word as part of his MS project. The University of North Florida's Board of Trustees has registered a copyright in connection with this project.

The authors would also like to thank the anonymous reviewers of this paper for their valuable comments.

9. REFERENCES

[1] Bruney, A. 2006. Professional VSTO 2005: Visual Studio 2005 Tools for Office (Programmer to Programmer). Wrox.

[2] Carter, E., Lippert, E. 2006. Visual Studio Tools for Office: Using C# with Excel, Word, Outlook, and InfoPath (First Edition). Addison-Wesley. Note: A second edition of this book will be published in 2009.

[3] Clements, P., Northrop, L. 2002. Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley.

[4] Gomaa, H. 2004. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley.

[5] Larman, C. 2004. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition). Prentice-Hall.

[6] Rumbaugh, J., Jacobson, I. and Booch, G. 2005. The Unified Modeling Language User Guide (2nd Edition). Addison-Wesley Professional.

[10] We are referring here to the books "Ontology Learning and Population from Text", by Philipp Cimiano (Springer); and "Ontology Learning from Text: Methods, Evaluations, and Applications", edited by Paul Buitelaar, Philipp Cimiano, and Bernardo Magnini (IOS Press). We look forward to reading the proceedings of the 3rd Workshop on Ontology Learning and Population, held in July of this year.

A Common Meta-Model for Data Analysis based on DSM

Yvette Teiken Stefan Flöring
OFFIS - R&D Division Health
Escherweg 2
26121 Oldenburg, Germany
teiken@offis.de floering@offis.de

Abstract

Realization and usage of advanced decision support systems are cost intensive. They require expert users during data collection and analysis task. Implementing these systems is time consuming and thus costly. This leads to the problem that SMEs (Small and Medium-sized Enterprises) often cannot afford these systems. In this paper we describe our aim of creating a DSM (Domain Specific Modeling) based top-down approach to generate advanced decision support systems. This approach is based on a family of DSLs (Domain Specific Language) that share a common meta-model. With this we aim to establish a faster and more affordable process for data analysis which better fits for SMEs.

Keywords DSM, DSL, Data Analysis, Visual Analytics, Decision Support Systems

1. Introduction

In our research group Data Management and Analysis we are dealing with the subjects collection, storage and analysis of complex multidimensional data. For this purpose we developed MUSTANG (Multidimensional Statistical Data Analysis Engine), a platform to implement specialized analytical information systems based on a data warehouse (Koch et al. 2003). It is, for example, used in the epidemiological cancer registry of Lower Saxony (Rohde and Meister 2004). The MUSTANG platform supports the collection and analysis of multidimensional data to supply information and decision support.

In this paper we describe a new top-down approach to data analysis based on DSM to extend the MUSTANG platform.

2. Brief overview of our research activities.

Available platforms for data analysis are often not optimized for specific domains or analysis approaches. Instead, they supply general solutions which require the work of a domain expert who establishes the analysis environment. Besides, the expert tries to find a visualization method which will fit the given analysis purpose best.

From our perspective this approach has two major shortcomings. First of all it requires that a large amount of visualization methods are integrated into the analysis platform. While this may technically not be challenging, it requires the data analysis expert to have an overview over a possibly large amount of visualization methods, of which many might not fit his needs. Another shortcoming results out of the data model which is used by the analysis platform. Multi-dimensional data models reclining the OLAP data model are typically used. This approach is of practical use because of the wide spread usage of systems which can provide data in this kind of model.

However, the allocation of indicators, derived from the multidimensional data model, to axis or other characteristics of a given visualization-method is not trivial. It requires deep semantic knowledge of the data. Data analysis therefore is a task limited to those users who exhibit this knowledge.

Both challenges lead to the conclusion that only a specialized data analyst with deep knowledge of both, the analysis domain and the technical characteristics of the visualization, can accomplish analysis tasks in an analysis platform like this. Reports generated by the analysis platform are then used as decision guidance. Therefore, the whole analysis process typically involves multiple users: data analysis experts and decision-makers. This procedure fits the needs of larger enterprises where data analysis and decision making are generally shared between different compartments due to the corporate hierarchy.

SMEs often do not distinguish between management-level and expert-level. Additional man power for analysis tasks is considered to be too cost intensive. This occurs in particular in healthcare environments which are under considerable cost-pressure. Nonetheless, there are vast amounts of data recorded in many healthcare applications which are a valuable source for data analysis. A streamlined and more cost efficient warehousing process which requires less knowledge by the analyst might help to establish warehousing in SMEs.

2.1 Model Driven MUSTANG

More and more organizations are using decision support systems like performance management (Friedrich 2007). Some of these approaches, e.g. Analytical Performance Management (Koch 2008), use a data warehouse (DWH) for data storage. A DWH is a central data storage from different sources. The data is used for data analysis for supporting decisions in organizations. The classical DWH process is a data driven approach. Analysis is based on existing data (Mucksch 2006). For supporting management decisions a demand driven approach is requested (Martin and Nußdorfer 2007). In a demand-driven process a decision maker formulates questions which shall be answered by the decision support system.

For integrating this kind of process in DWH a top-down approach must be realized. Therefore, we are creating a demand-driven top-down approach in MUSTANG. To accomplish this goal we want to define multiple DSLs as done in (Warmer and Kleppe 2006). A schematic diagram for our process is given in figure 1. This shows how an information demand in the context of epidemiology will be modeled and conditioned in our approach. In our approach a domain expert will be able to model relevant issues, his information demand, in a Measure DSL. These issues are expressed in key figures so the domain of this DSL is modeling key figures. The DSL will be used to model and as far as possible generate OLAP cube models. An OLAP cube is used to store a set of measures with dimensional context (Codd et al. 1993) and is used

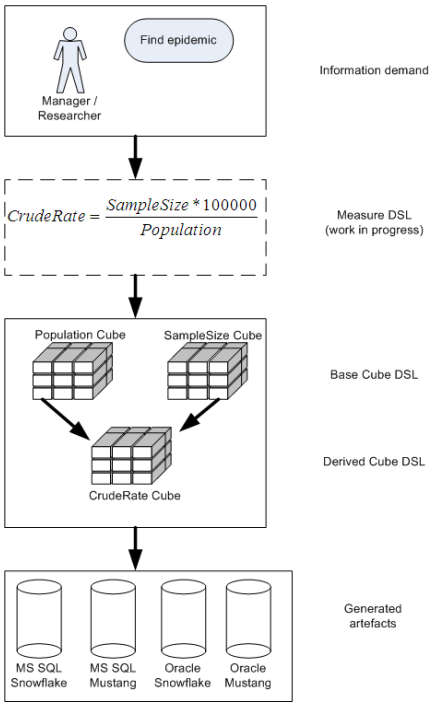


Figure 1. Process for data management in Mustang

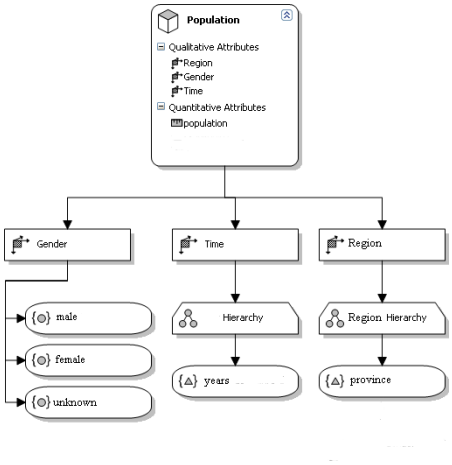


Figure 2. Example instance of our cube DSL

for data management. These models are multidimensional views on key figures with different levels of aggregation and are based on OLAP cubes. The cube itself is described by a cube DSL. In regard to the cube model the cube DSL is a more technical view on the domain and will be used as supplement of the Measure DSL. For better reuse and to benefit from a well-known language the DSL is based on ADAPT (Bulos 1996). Measures and dimensions are described graphically in the DSL. An example for a cube modeled in our cube DSL is given in figure 2. Based on this model we can generate a multidimensional data schema represented in SQL scripts. By means of a DSM approach we are able to support various databases and multidimensional schemata and applications for data integration based on the cube model. We can generate different kinds of multidimensional schemata like snowflake-, star- and MUSTANG-Schemata based on a cube-model. Supported databases are currently MS SQL and Oracle. This variety of data models is conditioned by the heterogeneous environments of our different research partners.

In future research we want to add software management tools like SAMA as described in (Koch and Teiken 2008). SAMA is a DSL based approach of monitoring strategy maps (Teiken 2008). We also want to add other DSLs to enable full generation of decision support systems and DWH. This will include a DSM based approach for security, integration and data quality.

2.2 Visual MUSTANG

Model driven MUSTANG is a first step for creating a streamlined, cost efficient data analysis solution. However, an approach for visualization is missing. We argued that domain experts with deep knowledge about both, visualization characteristics and data semantics, are required to choose appropriate visualization method. Our idea to bear this challenge is to use a visualization DSL to describe certain aspects of visualization methods, such as available dimensions or available operators on the visualized data. In (Hanrahan 2006) a language for this purpose, VizQl (visualization query language) is introduced. VizQl allows declarative queries for visualizations in a similar manner as SQL for data. We use this description to generate task or domain specific visualization applications which fit best for a certain analysis purpose.

We identified two steps in this process. One step which is technically motivated and domain independent and another step which is knowledge based and requires a domain expert. The first step is to identify technical characteristics of visualization models like the minimum and maximum number of dimensions and the available operators a certain visualization method provides. This is a technical task, which will result in a description language for the characteristics of visualization methods. In addition to this we want to be able to describe the explorative operators supported by a visualization method for a certain characteristic. In a map diagram, for example, a zoom-in operation for the visualization can intuitionally be used as drill-down operation.

The second step is more sophisticated. Here the knowledge of a domain expert about which kind of visualization is a good (or the best) choice for certain parameters of a given data model is needed. For example for one analysis task values with time dimension may be best visualized as an animation of diagrams. An example for this is the GapMinder [1]. Another task might require displaying time as index on the x-axis of a coordinate system.

Those two descriptions form a visualization DSL, which allows to describe the visualization methods best fitting to certain data model or analysis task. A similar DSL for data models is necessary to create a software development process which will allow automatic generation of visualization applications for certain data models. A similar approach is used in (Bull 2006) where model driven visualization is used to rapidly prototype new visualizations.

3. A Common Meta-Model for data analysis

In our joint work we noticed that many of these research questions may be solved with the help of a DSM based approach based on a common meta-model as recommend in (Hessellund et al. 2007). Our different approaches will be realized by using different DSLs. We call those related DSLs a family of DSLs. We aim for our family of DSLs to share a common meta-model. The reason why we use different DSLs rather than different view points is variousness of our reception radius. We address our DSLs to different kinds of domain experts like a manager or security expert. In case of cube and dimension modeling it would be appropriate to use view points.

[1] http://www.gapminder.org

From this common model we expect synergies in the development of a streamline process for all phases in data analysis, and thus a more cost effective realization of projects which will allow SMEs to take advantage of data analysis.

For every DSL a meta model to express its abstract syntax is essential (Völter and Stahl 2006). For the integration of our DSLs we decided to use a common meta-model. A simplified version of the MUSTANG DSL meta-model is shown in figure 3. The cube DSL is connected to the Measure DSL via measure. A cube can store a single measure in different granularities. This is called a base measure. A measure can also be a combination of other measures which is called a derived measure. Each cube has a number of operations and dimensions with hierarchies. An operation describes potential OLAP operations for a cube. The dimension has a type that describes special properties, e.g. if it's a geographical dimension. An example of a simplified instance of our meta-model is given in figure 4. It shows how the information demand "find epidemic" in figure 1 with help of crude rate is expressed. One application of these properties can be the type of visualization of the dimension.

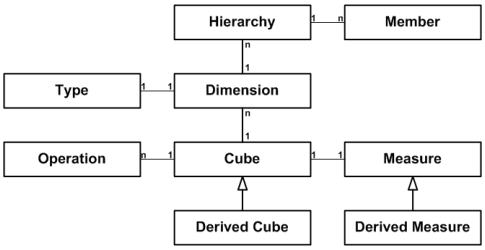


Figure 3. Part of MUSTANG Meta-Model

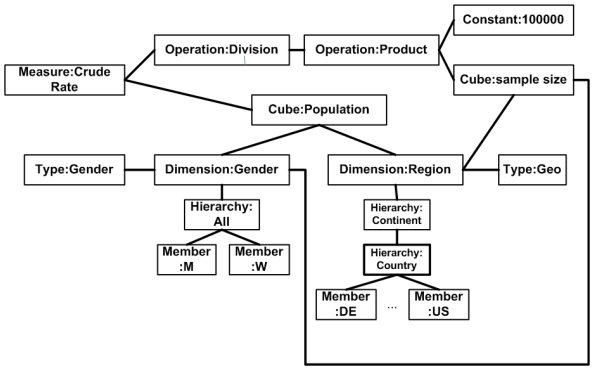


Figure 4. Instance of our meta-model

A meta-model for the visualization DSL needs to include the following two items: presentable characteristics of the visualization method (dimensions, hierarchies, members) with value margins and operators supported by the visualization. In our research we discovered that the meta-model which is used for the cube DSL can be used for the visualization DSL as well if the annotations mentioned are added. Therefore, we decided to use a common meta-model for both languages.

Figure 5 shows how visualizations described by the visualization DSL are matched to an instance of the MUSTANG meta-model. At first the characteristics of the data model are detected. After this step the amount of available visualizations is limited to those which support the characteristics of the data model. In the next step annotations of the data model which are derived from expert knowledge are evaluated. In the described process of generating a top-down decision support system the model incorporates knowledge by expert users. This knowledge is available to the matching process. This matching process results in the best matching visualization for the application generation. If more than one visualization is applicable to a certain task, there are two choices: Either an expert user can manually influence the generation process and choose one of these visualizations or an application with the support of multiple visualizations will be generated.

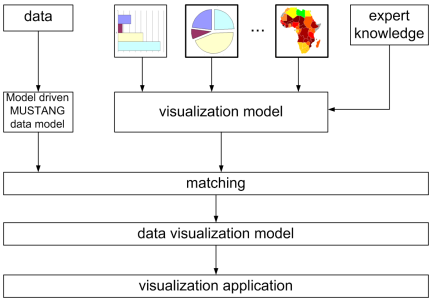


Figure 5. Visualization Matching Process

4. Conclusion

Using the DSM based approach a decision support system with appropriate visualization can be realized cost efficient. With our family of DSLs we can generate suitable applications based on models described by domain experts with a minimum of manual programming.

Another advantage of this approach is a possible higher user satisfaction. When modeling measures and cubes with the specific DSLs we generate domain knowledge. This domain knowledge can be used to choose not only a technically fitting but a semantically appropriate visualization.

On the other hand due to the DSM approach it is easier to create and integrate additional visualizations into MUSTANG. The characteristics of a visualization method may be described and applications can be adapted without implementation work afterwards. Using the common meta-model the integration of data can be fully defined within the model, reducing the developers work for data integration. These two aspects open the perspective for a DSM based roundtrip-engineering (Antkiewicz and Czarnecki 2006) where exploration and visualization of data might imply adjustments to the model which with help of our family of DSLs may automatically be propagated to the model.

References

Michal Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 692–706. Springer, 2006. ISBN 3-540-45772-0.

R. Ian Bull. Integrating dynamic views using model driven development. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, page 17, New York, NY, USA, 2006. ACM. doi: http://doi.acm.org/10.1145/1188966.1188989.

Dan Bulos. Olap database design: A new dimension. *Database Programming&Design*, Vol. 9(6), 1996.

Edgar F. Codd, Sharon B. Codd, and Clynch T. Salley. Providing olap to user-analysts : An it mandate. White paper, E.F. Codd Associates, 1993.

Dirk Friedrich. Einfach soll es sein - bei hoher datenqualität. *IS - Informationsplattform für Business Applications*, 11:30–35, 2007. (in german).

Pat Hanrahan. Vizql: a language for query, analysis and visualization. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 721–721,

New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. doi: http://doi.acm.org/10.1145/1142473.1142560.

Anders Hessellund, Krzysztof Czarnecki, and Andrzej Wasowski. Guided development with multiple domain-specific languages. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2007. ISBN 978-3-540-75208-0.

Sascha Koch. *Analytisches Performance Management*. PhD thesis, Universität Oldenburg, 2008. (in german).

Sascha Koch and Yvette Teiken. Semi-automatische überwachung von zielsystemen. In Martin Bichler, Thomas Hess, Helmut Krcmar, Ulrike Lechner, Florian Matthes, Arnold Picot, Benjamin Speitkamp, and Petra Wolf, editors, *Multikonferenz Wirtschaftsinformatik*. GITO-Verlag, Berlin, 2008. ISBN 978-3-940019-34-9. (in german).

Sascha Koch, Jürgen Meister, and Martin Rohde. MUSTANG – A Framework for Statistical Analyses of Multidimensional Data in Public Health. In *Proceedings of the 17th International Conference Informatics for Environmental Protection*, pages 635–642, Cottbus, September 2003. (in german).

Wolfgang Martin and Richard Nußdorfer. Cpm – corporate performance management, kompendium:analytische services in einer soa, teil 1: Herstellerunabhängige beschreibung und referenzarchitektur. White Paper, August 2007. (in german).

Harry Mucksch. *Analytische Informationssysteme. Business Intelligence-Technologien und -Anwendungen: Business Intelligence-Technologien Und -Anwendungen*, chapter Das Data Warehouse als Datenbasis analytischer Informationssysteme, pages 129–142. Springer, Berlin, 2006. (in german).

Martin Rohde and Jürgen Meister. *Data-Warehouse-Systeme. Architektur, Entwicklung, Anwendung*, chapter Data Warehousing in der Gesundheitsberichterstattung, pages 484–498. dpunkt Verlag, 2004. (in german).

Yvette Teiken. *Semi-automatische Überwachung Annotierter Strategy Maps*. 2008. ISBN 3899636724. (in german).

Markus Völter and Thomas Stahl. *Model-Driven Software Development*. Wiley & Sons, 2006.

J. B. Warmer and A. G. Kleppe. Building a flexible software factory using partial domain specific models. In *Sixth OOPSLA Workshop on Domain-Specific Modeling (DSM'06), Portland, Oregon, USA*, pages 15–22, Jyvaskyla, October 2006. University of Jyvaskyla. ISBN 951-39-2631-1.

Towards Model-Based Testing of Domain-Specific Modelling Languages

Janne Merilinna
VTT Technical Research Centre of Finland
P.O. Box 1000, 02044 Espoo, Finland
janne.merilinna@vtt.fi

Olli-Pekka Puolitaival
VTT Technical Research Centre of Finland
P.O. Box 1100, 90571 Oulu, Finland
olli-pekka.puolitaival@vtt.fi

Juha Pärssinen
VTT Technical Research Centre of Finland
P.O. Box 1000, 02044 Espoo, Finland
juha.parssinen@vtt.fi

Abstract

Domain-Specific Modelling (DSM) has evidently increased the productivity and the quality of software development. The witnessed gains are primarily caused by the three corner stones of DSM, i.e. Domain-Specific Modelling Languages (DSML), code generators and software frameworks. Although the DSMLs and the code generators are the primary reason for the gains, little attention have been paid in making sure that these work correctly. In this paper, we present a work in progress on the technique of utilizing the Model-Based-Testing (MBT) as a means for testing the elements of the DSM basic architecture. We will discuss how the MBT can be utilized for generating a comprehensive test suite of application models, in addition to how the generated applications can be tested with the MBT. As a combination, the DSM basic architecture will be tested thoroughly. We also present how the introduced technique can be realized by utilizing the tools currently available for the DSM and the MBT.

Categories and Subject Descriptors D.2.5 [**Testing and Debugging**]: Testing tools

General Terms Languages, Verification.

Keywords Metamodel testing

1. Introduction

Domain-Specific Modelling (DSM) is all about raising the level of abstraction from thinking of the software in a solution-space into a level concerning the software in a problem-space. Although, during research, there are no or only a few tiny experiments [1] in comparing the benefits of DSM and traditional software development means, cases conducted in industrial settings constantly show 5-10 times productivity gains compared to the traditional software development [2,3]. In addition to the productivity gains, the DSM is expected to have a positive impact on the software quality [3].

Productivity and quality gains are primarily caused by the three corner stones of DSM, i.e. Domain-Specific Modelling Languages (DSML), code generators and software frameworks which when combined are known as the DSM basic architecture [3]. Instead of modelling the software in solution-space, DSMLs provide concepts that directly map the concepts found in the problem domain. These concepts also include, in addition to elements found in the domain, restrictions that guide the application developer in developing applications that function correctly. The responsibility of the code generator is then to transform these high-level specifications into a source code running on top of the target platform. By automating the model to source code transformation, a great deal of error-prone translations that otherwise had to be implemented manually can be omitted.

Testing in a traditional software development is one of the corner stones in order to raise quality. Regardless of the fact that it is the DSML and its code generator that are the primary sources of errors, there are not that many publications that consider testing metamodels and code generators. In [4], Sadilek and Weißleder present a technique for testing metamodels. These models are comprised of positive and negative test models that are utilized for evaluating the correctness of the metamodel. The positive test models are comprised of models that should be possible to be modelled by using the metamodel, where the negative test models are something that shouldn't be possible to model. Stuermer et al. in [5] present a systematic code generator testing technique, which is applicable in situations where the source and target languages are executable. This essentially means the comparison of the behaviour of a model that can be simulated for an executable run in a target environment. In [3], Kelly and Tolvanen consider testing DSMLs and code generators as a combination. They suggest that the language creation should be considered incremental and test case driven. This means the development of the language and the code generator in small incremental steps and testing these by developing some small applications or by rebuilding applications that have already been developed. In this way, it is expected that the DSM basic architecture will be tested thoroughly.

In situations where one can not afford to release the language and its supporting code generator prior to making sure that they function correctly, not enough confidence can be attained without systematic testing. Particularly, in the case where software development is iterative and incremental, a question whether the languages and code generators under continuous change still function correctly in all cases is raised. In addition, there is an issue of test suite maintenance when metamodels, code generators and application models evolve.

Model-based testing (MBT) [6] is a prominent black box software testing method that enables the creation of a comprehensive test suite by modelling the behaviour of a system, where the models can then be transformed into a test suite by utilizing several test design algorithms [7]. This enables automating the generation of a test suite from models that are easier to keep in synch with the evolving software system. As the test suites are based on models, the maintenance effort of the test suite also decreases.

In this paper, we present a work in progress on applying the MBT for testing the DSM basic architecture in the context of an

iterative and incremental software development process. We also discuss the topic from the view point of how the testing technique can be realized by utilizing the existing tools for the MBT and the DSM.

This paper is structured as follows. First, the basic principles of MBT are briefly discussed. Second, the technique for testing the DSM basic architecture is presented. Third, how the technique can be realized in practice is presented, followed by a discussion and conclusions. The final remarks close the paper.

2. Model-Based Testing

In the past, regression tests for test automation systems have been developed manually [8]. When implemented manually, there is always an extra effort in maintaining the test scripts. So-called keyword-driven testing [9] has been seen as a prominent method for solving the test script maintenance issue. However, keyword-driven testing still requires manual implementation, therefore the problem is only partially solved. MBT is seen as a complementary approach for solving the test script maintenance issue by automating designing and implementation of the test suite [6].

The MBT is a black box software testing method in which the test scripts are automatically generated from a model which describes the behaviour of the system under test (SUT) [10]. The test scripts are generated from a model by utilizing a set of test design algorithms [7] that traverse the model and generate test scripts from that basis.

The MBT process can be divided into three phases, i.e. modelling, test generation and test execution (Figure 1). The modelling stands for the modelling of system behaviour, where the test generator then generates a test suite from these models. The test executor then conducts the test. Next in this section, the phases of the MBT are discussed briefly.

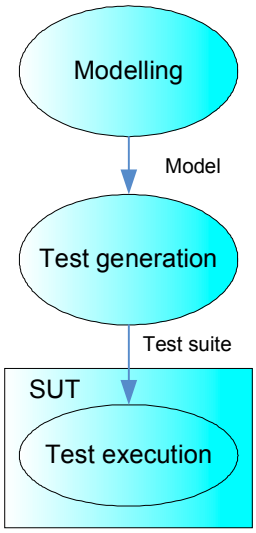


Figure 1. Model-based testing process.

2.1 Modelling

The functional requirements of the software systems is the primary source for developing MBT models [6 p.27]. These models embody the externally visible behaviour of the system. As the system requirements are also modelled with the MBT, in addition to realizing the requirements as an implementation, exist two opinions of the behaviour of the system. The differences between these can be viewed as errors.

The model is required to have knowledge of the input and output data of the SUT. The input data is used for executing the tests and the output data is for validating the tests. The model can be made from an environmental [11] or design [12] viewpoint. The viewpoints are mirror images and are equally suitable for test generation. The design viewpoint for the MBT is similar to the modelling viewpoint for implementation purposes, but at a higher abstraction level. The implementation model can therefore be reused to model design viewpoint MBT model [6]. The notation of the models can be graphical, textual or mixed, where the notation varies from general purpose to domain-specific [10].

2.2 Test Generation

Test generation is based on a model traversal, where several algorithms, called test design algorithms, are utilized for generating test cases from the model. There are three main categories of test design algorithms [13]:

- Requirement-based criteria i.e. the test generator strives to cover all the marked requirements,
- Coverage criteria, i.e. a test suite is generated on the basis of covering a certain degree of the model, and
- Walking algorithms, i.e. an algorithm determines how the model is traversed and the test suite is generated from that basis.

2.3 Test Execution

A test execution can be performed either offline or online [10]. The offline testing stands for generating tests first and then executing the tests separately, whereas with online testing, one step at a time is generated based on the output of the SUT. The difference between the test approaches is that in the offline approach, test generation is separated from test execution and therefore there is a possibility to utilize algorithms requiring heavy computation without the test execution suffering. In the case of online testing, the following test step is generated on the basis of the previous step output values of the SUT, therefore in order to have an immediate response to outputs, only algorithms requiring less computation can be utilized. Online testing also enables the ability to have infinite test suites and enables handling the non-deterministic behaviour [14] of the SUT.

3. Technique for Testing the DSM Basic Architecture by Utilizing Model-Based Testing

Software testing, in the case of DSM, essentially means testing the primary sources of errors, i.e. testing the metamodel and the code generator. Although being the primary sources of errors, not much research have taken place to address these [3, 4, 5].

In the context of iterative and incremental software development processes, there is also concern about the test maintenance because the applications evolve. Additionally, code generators are under constant evolution, when the underlying platform evolves. Changes in the metamodel will also have an impact on the code generators, whereas changes in the metamodel and the code generators have an impact on the existing applications. When one of the three aspects evolve, one has to therefore have tests in order to make sure that the evolved versions function correctly.

In this paper, it is argued that testing application models, metamodels and code generators cannot be performed separately since all of the three levels are intertwined tightly together. This is because modelling languages consist of syntax and semantics definitions. The metamodel describes the semantics of the model, but it cannot have an impact on how the code generator decides to produce the code. It is therefore the code generator that makes the ultimate decision for what is generated and how and thus the metamodel and the code generator, as a combination, define the semantics in practice. This speaks on behalf of the testing practice introduced in [3]. However, the presented approach does not take into account that the code generator and the metamodel cannot be tested thoroughly by only a couple of example applications.

MBT can be seen as a prominent method for generating comprehensive test suites. This statement is based on the capability of the test design algorithms to produce a comprehensive test suite from the MBT models. A model being easier to maintain compared to a test suite, decreases the maintenance effort [6]. In addition to lightening maintenance, the test suite will always be in synch with the MBT model. Next in this section, techniques for utilizing the MBT in testing the DSM basic architecture is discussed.

3.1 Generating a Test Case for the DSM Basic Architecture

The MBT can be utilized for testing the applications developed with the DSM approach. However, testing the applications differs in the case of DSM, from the traditional MBT. In the traditional MBT, models and the implementation are derived from informal software specifications, therefore it is tested whether the application implementation follows the specifications. In the DSM approach, the implementation and the MBT are derived from the same model, thus it cannot be tested whether the application is implemented according to the specifications but it is tested whether the code generator produces a working application running on the software framework from the application model. Thus, in the context of DSM, one application model can be seen as a test case for the whole DSM basic architecture.

The technique for utilizing the MBT in developing a test case for the DSM basic architecture is illustrated in Figure 2. The left side of the figure follows the basic code generation process, whereas the right side follows the MBT process. Considering the MBT process, the difference between the introduction of the DSM to the MBT and the traditional MBT is in the source of the model. Whereas in the traditional MBT the model is derived from software specifications, in this case the model is derived from the same model that the DSM utilizes for the code generation. By doing so, both the code generator and the MBT tool always have the same conception of the model. Errors are detected if the code generator realizes the conception differently, thus the MBT model no longer matches the generated application when executed.

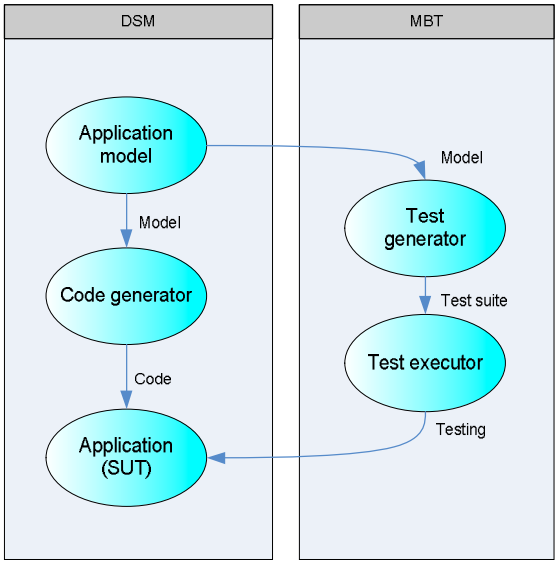


Figure 2. MBT utilized for testing applications.

3.2 Generating a Test Suite for the DSM Basic Architecture

One test case does not test the whole DSM basic architecture thoroughly. By modelling of a set of applications, test coverage increases. However, the effort to maintain the test suite, i.e. a set of application models, can become an issue when the DSM basic architecture evolves.

As in the case of the DSM, all applications of a certain domain are based on a metamodel, the test suite for the DSM basic architecture should also be based on this metamodel. In this paper, it is argued that the MBT can be utilized for generating a comprehensive test suite of application models for the DSM basic architecture from a metamodel. This claim is based on the capability of the MBT to generate a test suite from models, of which metamodels essentially are. If the test suite is generated from a metamodel, it will also always be in synch with the evolving language, thus decreasing the maintenance effort of the test suite.

An overview of the technique for generating a test suite for the DSM basic architecture is depicted in Figure 3. It must be noted that it is the application models that are generated from the metamodel by the MBT tool. As the MBT tool utilizes the metamodel for generating the test suite, the metamodel has to be strictly defined, i.e. the metamodel should be defined in such a way that only legal applications can be modelled. If not, the metamodel test suite also includes test cases, i.e. application models, that are not legal for the target platform. With enough application models, the errors in the DSM basic architecture can be noticed.

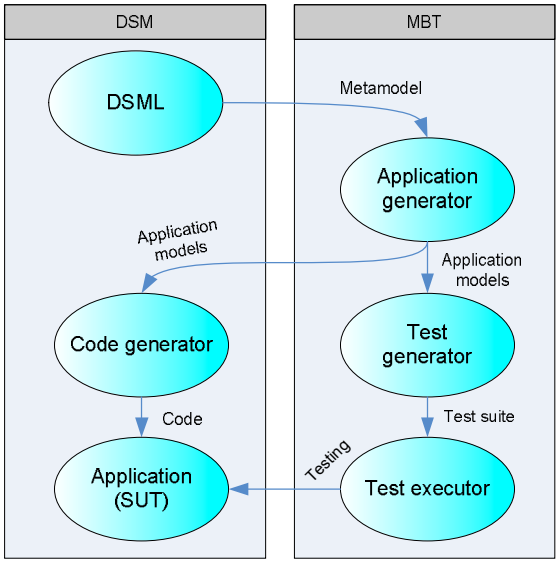


Figure 3. Testing the metamodel.

4. Test Suite Generation for the DSM Basic Architecture – Illustration with the Existing Tool Support

The technique for generating a comprehensive test suite for the DSM basic architecture requires an extensive tool support. The DSM requires a language workbench that provides modelling, metamodelling and code generation facilities, whereas the MBT requires an environment that takes, as an input, a model that is utilized for the test suite generation. Additionally, an extensive collaboration between the tools is also required. The way the tools can collaborate has an impact on how the test suite generation can be realized in practice. Thus, in this section, the test suite generation technique is illustrated from the view point of the currently available tool support. First, currently available tool support for the MBT and the DSM are discussed and two of the most suitable tools have been chosen for this illustration. Second, the technique is discussed from the view point of the selected tools.

4.1 Tools for Domain-Specific Modelling and Model-Based Testing

Enabling the test suite generation requires a DSM tool providing facilities for exporting the metamodel into a format required by the MBT tool. In addition, the DSM tool has to enable importing models generated by the MBT tool, whereas the MBT tool has to enable exportation of the test suite, i.e. a set of application models, to the DSM tool, in addition to providing facilities for generating the test suite for the imported application models.

4.1.1 Tools for the Model-Based Testing

Tool support for the MBT is extensive [6, p.401-403] [15]. However, many of the MBT tools are still immature but there also exists commercial tool vendors providing more mature tools and support when required. MaTeLo from All4Tec[1] is for control oriented MBT. Reactis from Reactive Systems[2] and T-Vec[3] provides tools focused in embedded MBT testing. Test Designer from Smartesting[4] and Conformiq Qtronic from Conformiq[5] are general-purpose solutions for the MBT. Scrutinizing the MBT tool evaluation, presented in [13], reveals the Qtronic to be mature enough and provides open data formats for importing and exporting the models.

The Qtronic expects the input model to be either in

- a similar format as the UML state machine diagram extended with a variant of Java, which is called QML, or
- a textual representation, where the programming language is QML.

Both of the model types represent a model where the input and output pairs are defined. In addition, special requirements can be defined for the inputs and outputs that guide the test design algorithms in generating the test suite. The test suite generator can be implemented by utilizing the provided plug-in interface.

4.1.2 Tools for the Domain-Specific Modelling

The Generic Modeling Environment 6 from Vanderbilt University[6] and Metaedit+ 4.5 from Metacase[7] are probably the most well-known language workbenches. Microsoft also provides a DSM tool with Microsoft Visual Studio 2005 SDK[8]. There are also open source tools available, such as the Generic Eclipse Modeling System[9].

Metaedit+ is our choice among the tools since, as far as we know, it is the only language workbench providing code generator facilities with a language dedicated only for developing code generators. This enables a rapid development of generators compared to a situation where the code generators are developed by utilizing the APIs of the modelling tools. In addition, Metaedit+ enables importing models in an XML format.

4.2 Test Suite Generation in Practice

In order for Metaedit+ to export its metamodel to QTronic, a metamodel has to be modelled with the included GOPPRR modelling language. Exporting the metamodel requires a Metamodel-to-QML (Met2QML) code generator that takes a metamodel as an input and generates QML out of it. Now, the Qtronic imports the generated metamodel. By utilizing the means of the MBT, the Qtronic traverses the metamodel extensively and stores the traversed paths as lists of visited concepts. The traversed paths then form a test suite of application models. The test suite of application models is then utilized for generating application

[1] http://www.all4tec.net/
[2] http://www.reactive-systems.com/index.msp
[3] http://www.t-vec.com/solutions/products.php
[4] http://www.smartesting.com/
[5] http://www.conformiq.com/qtronic.php
[6] http://www.isis.vanderbilt.edu/projects/gme/
[7] http://www.metacase.com/mwb/
[8] http://msdn2.microsoft.com/fr-fr/vstudio/aa718368(en-us).aspx
[9] http://www.eclipse.org/gmt/gems/

models in a XML format required by the Metaedit+. The test suite of application models is then imported by the Metaedit+, which then generates a source code from the models. As a result, there is a set of applications ready to be executed and tested.

In order to test the generated applications, the application model of each application has to be imported back to the Qtronic. This requires a model-to-QML (Mod2QML) code generator that generates application models into a format required by the Qtronic. The Mod2QML now generates application models in a format required by the Qtronic, which then generates a test suite for each application. The test suites for applications are then forwarded to the test executor, which then conducts the test. An overview of the workflow is depicted in Figure 4.

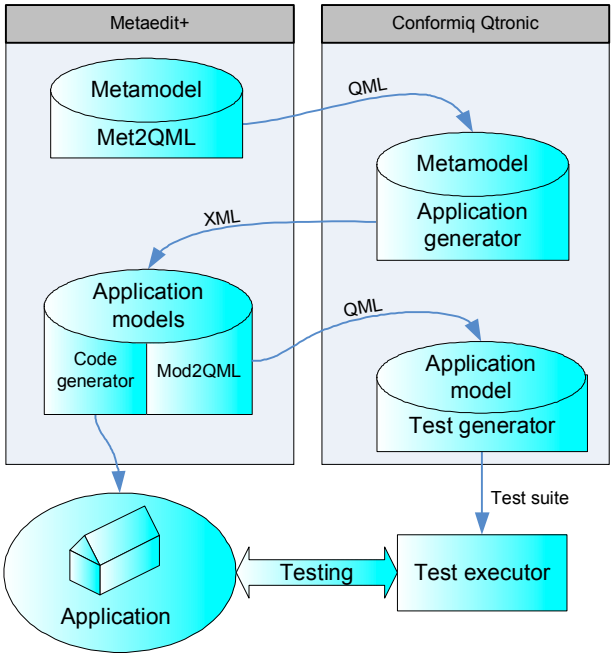


Figure 4. Overview of the workflow for generating a test suit for the DSM basic architecture.

5. Discussion

Test automation has decreased the cost of testing, but there is still an issue of test maintenance. MBT strives to ease the maintenance by test suite generation. A technique for generating a test suite for the DSM basic architecture merely brings another set of entities that have to be tested. This also causes, in addition to more things to test, more aspects to maintain. It is a justifiable question whether this worth for it. Currently, there is no answer to the question. It is not known how much extra effort it takes to have this test and what are the benefits of it compared to rather adhoc DSML testing. In addition to these, a few open questions exist.

- Can all metamodels be transformed into a format required by the MBT tools, i.e. are there any special cases where the metamodel cannot be transformed into a MBT model,
- is it feasible to generate application models from the metamodel by an approach of MBT,
- are there special cases where the MBT cannot be utilized for generating a test suite for the applications from the DSML models, and
- how to automate the whole process.

In order to deal with the open questions, we will continue the development of 1) a generator for generating a test suite from DSML models for applications and 2) a generator for generating a test suite of application models from the metamodel.

6. Conclusion

Industrial cases constantly reveal 5-10 productivity gains when utilizing DSMLs in the software development. Although the gains witnessed so far are great, a concerned question raised is how to make sure the languages, i.e. metamodels and code generators, are correct especially in the cases where the languages are provided for the masses and for mission critical systems. Iterative and incremental software development processes bring an additional question of test suite maintenance. Although testing is an important means to detect errors in the traditional software engineering, it is not well known how to test the DSM basic architecture thoroughly.

Our contribution is a technique for utilizing the MBT for generating a test suite for the DSM basic architecture. The presented technique does not strive for the testing of the layers of the DSM basic architecture in isolation, but to test the whole architecture. The test technique is based on generating a comprehensive test suite of application models by utilizing the means provided by the MBT plus generating test suites for the generated applications. As a combination, the DSM basic architecture will be tested thoroughly. The presented technique is also discussed from the tool support point of view, in order to enable further discussion on the feasibility of the introduced technique.

References

[1] Merilinna, J. and Pärssinen, J., "Comparison between different abstraction level programming: experiment definitions and initial results", The 7th OOPSLA Workshop on Domain-Specific Modeling, Montreal, Canada, 2007

[2] Hammond, J. L., "Domain-specific modeling significantly reduces development time", URL: http://www.metacase.com/papers/ece_april2008.pdf [Visited at 18.6.2008]

[3] Kelly, S. and Tolvanen, J-P, "Domain-Specific Modeling: Enabling full code generation", John Wiley & Sons, ISBN 978-0-0470-03666, 427 p., 2008

[4] Sadilek, D. A. and Weißleder, S., "Testing Metamodels", Fourth European Conference on Model Driven Architecture Fondations and Applications, Berlin, Germany, 2008

[5] Stuermer, I., Conrad, M., Doerr, H., and Pepper, P., "Systematic Testing of Model-Based Code Generators", IEEE Trans. Softw., pp. 622-634, 2007

[6] Utting, M. and Legeard, B., "Practical Model Based Testing: A Tools Approach", Morgan Kaufmann 1st ed., ISBN: 978-0123725011 , 456p., 2006

[7] Utting, M., Pretschner, A. and Legeard, B., "A taxonomy of model-based testing", Working papers series. University of Waikato, Department of Computer Science, Hamilton, New Zealand, University of Waitako, 2006

[8] Wahl, N., "An overview of regression testing", ACM SIGSOFT Software Engineering Notes, Volume 24 Issue 1, ISSN:0163-5948, pp. 69-73, 1999

[9] Mosley, D. and Posey, B., "Just Enough Software Test Automation", Yourdon Press. Prentice-Hall, ISBN:0130084689, 300p., 200

[10] Hartman, A., Katara, M. and Olvovsky, S., "Choosing a test modeling language: A survey", Haifa Verification Conference, pp. 204-218, 2006

[11] Auguston, M., Michael, J. and Shing, M., "Environment Behavior Models for Scenario Generation and Testing Automation", ACM SIGSOFT Software Engineering Notes, Volume 30 Issue 4, Advances in Model-Based Testing (A-MOST 2005), 2005

[12] Schulz, S., Honkola, J. and Huima, A., "Towards Model-Based Testing with Architecture Models", Engineering of Computer Based Systems (ECBS '07), pp. 495-502, 2007

[13] Puolitaival, O.-P., Luo, M. and Kanstren, T., "On the Properties and Selection of Model-Based Testing tool and Technique", 1st Workshop on Model-based Testing in Practice (MoTiP 2008), Berlin, Germany, 2008

[14] Nachmanson, L., Veanes, M., Schulte, W., Tillmann, N. and Grieskamp, W., "Optimal strategies for testing nondeterministic systems", ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 55–64, 2004

[15] Hartman, A., "Model Based Test Generation Tools", Agedis Consortium, URL: http://www.agedis.de/documents/ModelBasedTestGenerationTools_cs.pdf, 2002

MODELTALK: A Framework for Developing Domain Specific Executable Models *

Atzmon Hen-Tov
Pontis Ltd.
Glil Yam 46905, Israel
atzmon@pontis.com

David H. Lorenz
The Open University of Israel
108 Ravutski St., Raanana 43107, Israel
lorenz@openu.ac.il

Lior Schachter
Pontis Ltd.
Glil Yam 46905, Israel
liors@pontis.com

Abstract

Developing and maintaining complex, large-scale, product line of highly customized software systems is difficult and costly. Part of the difficulty is due to the need to communicate business knowledge between domain experts and application programmers. Domain specific model driven development (MDD) addresses this difficulty by providing domain experts and developers with domain specific abstractions for communicating designs. Most MDD implementations take a generative approach. In contrast, we adopt an interpretive approach to domain specific model driven development. We present a framework, named MODELTALK, that integrates MDD, dependency injection and meta-modeling to form an interpretive, domain specific modeling framework. The framework is complemented by tool support that provides developers with the same advanced level of usability for modeling as they are accustomed to in programming environments. MODELTALK is used in a commercial setting for developing a product line of Telco grade business support systems (BSS).

Categories and Subject Descriptors D2.6 [*Programming Environments*]: Programmer workbench; D3.2 [*Language Classifications*]: Extensible languages, Object-oriented languages

General Terms Design, Languages

Keywords Model driven development; Dependency injection; Meta-modeling; Executable model; Domain specific languages

1. Introduction

Modern business application development is complex. It involves several domains of expertise, dealing with both functional and extra-functional requirements, all complicating the communication between domain users and domain experts. Working with domain specific models alleviates some of this complexity by communicating domain abstractions in designs.

In this work, we present a framework, named MODELTALK, for developing domain specific executable models. An *executable model* [17, 12] is a model that drives the execution of the system. The major virtue of an executable model is that changes in the model are automatically reflected in the system [24]. MODELTALK is an interpretive, domain specific modeling framework: the model is the primary source of the system; the desired behavior of the runtime system is achieved by interpreting the model.

MODELTALK integrates the principle of domain driven development [22, 19] with the technique of dependency injection. *Dependency injection* [15, 10] is a mechanism for defining external dependency declaratively (e.g., as an object graph configuration in XML) that can be injected into the runtime system (e.g., into Java objects). The major virtue of dependency injection is that it supports declarative changes. The system behavior can be modified by composing descriptions of object graphs (in XML), thus avoiding the long cycle of compile-pack-deploy that is required when the changes are done in code (in Java).

The MODELTALK framework is complemented by tool support [11]. An Eclipse [8] plug-in for MODELTALK provides developers with the same advanced level of environment look-and-feel for modeling as they are accustomed to with programming.

Outline The rest of the paper is structured as follows. Section 2 briefly reviews dependency injection by example, comparing a code driven to a model driven approach. In Section 3 we describe the high level architecture of MODELTALK and show how modeling and coding are integrated to form a model driven development framework. In Section 4 we illustrate meta-modeling with MODELTALK. Assessment of the MODELTALK framework is brought in Section 5.

2. Model Driven Dependency Injection

In this section, we illustrate the concept of code driven dependency injection in the Spring [21] framework. We then contrast dependency injection in Spring with the concept of model driven dependency injection in MODELTALK.

2.1 Code Driven Dependency Injection in Spring

In Spring, the developer starts the development iteration cycle by working on the Java implementation. Instances (*beans*, in Spring's terminology) are then defined to customize the implementation. As an example, consider the UML domain model for an HTTP client system depicted in Figure 1.[1] The Java class `HTTP_Client` (Listing 1) provides a `sendReceive` method for sending HTTP requests. The class has three private instance variables: `numberOfRetries` and `timeout` are used for configuring its communication handling policy; `URL` is used for configuring the Internet address of the resource to be accessed.

An XML bean in Spring is a description of an object graph. It is instantiated into Java objects at runtime. The XML excerpt in Listing 2 shows how one might use beans in Spring to customize the `HTTP_Client` class:

1. `RobustHTTP_Client` defines an abstract instance [21] of `HTTP_Client` with high numerical values for `timeout` and for `numberOfRetries`.

* This research was supported in part by the *Israel Science Foundation (ISF)* under grant No. 926/08 and by the office of the chief scientist of the Israel Ministry of Industry Trade and Labor.

[1] The UML diagrams are for illustration only. Models in MODELTALK are expressed in XML.

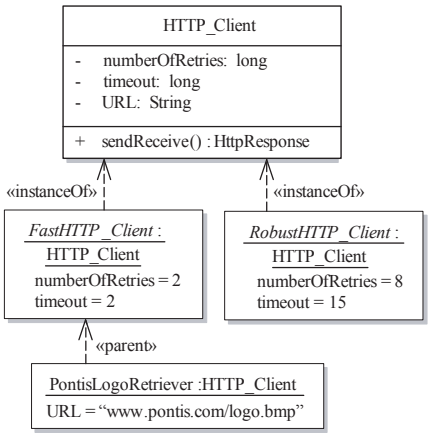


Figure 1. Domain model

```
public class HTTP_Client {
  private long numberOfRetries = 0;
  private long timeout = 0;
  private String URL = null;

  public void setNumberOfRetries(long number) {
    this.numberOfRetries = number;
  }
  public void setTimeout(long timeout) {
    this.timeout = timeout;
  }
  public void setURL(String URL) {
    this.URL = URL;
  }
  public HttpResponse sendRecieve() {
    HttpResponse result = null;
    //business logic
    return result;
  }
}
```

Listing 1. Class implementation in Java

```
<bean id="RobustHTTP_Client" class="HTTP_Client"
      abstract="true">
  <property name="numberOfRetries" value="8"/>
  <property name="timeout" value="15"/>
</bean>
<bean id="FastHTTP_Client" class="HTTP_Client"
      abstract="true">
  <property name="numberOfRetries" value="2"/>
  <property name="timeout" value="2"/>
</bean>
<bean id="PontisLogoRetriever" class="HTTP_Client"
      parent="FastHTTP_Client">
  <property name="URL"
            value="www.pontis.com/logo.bmp"/>
</bean>
```

Listing 2. XML beans in Spring

```
public static void main(String[] args)
    throws MalformedURLException {
  GenericApplicationContext context=getSpring();
  HTTP_Client httpClient =
    (HTTP_Client)context.getBean("PontisLogoRetriever");
  HttpResponse response = httpClient.sendReceive();
  // ...
}
```

Listing 3. Client code in Java

2. `FastHTTP_Client` defines an abstract instance of `HTTP_Client` with low numerical values for `timeout` and for `numberOfRetries`.
3. `PontisLogoRetriever` defines a *concrete* instance of `HTTP_Client` by specializing `FastHTTP_Client` with the location for the logo bitmap.

This form of customization works for simple as well as for arbitrary complex object graphs. For simplicity, the example illustrates the use of Spring for configuring properties of primitive types. Generally, however, the injected values may also be instances of user defined classes.

Lastly, the Java excerpt in Listing 3 shows how a client code uses the Spring factory to instantiate an `HTTP_Client` with the desired configuration.

2.2 Model Driven Dependency Injection in MODELTALK

In MODELTALK, the developer starts the development iteration cycle by working on the *model*. MODELTALK uses the notion of a class definition in the model. Model class definitions are the primary source in which the constraints for the XML beans and for the structure of the implementation code are defined. These constraints are reflected in the development tool immediately, providing the developer with full support for auto-completion, consistency checking, and so on.

The XML excerpt in Listing 4 is the model class definition of `HTTP_Client` (and its three model instances) in MODELTALK.[2] The definition informs the modeling tool about the existence of this class. This is in contrast to Spring, where one must have the Java class itself available. MODELTALK uses property name tags to provide domain specific syntax.

Model driven dependency injection enhances the safety of the declarative change process. Class definitions in the model constrain the model objects, leading to early detection of errors. Changes to the model can be applied to the runtime system with a higher degree of confidence than in Spring, since they undergo consistency checking.

In the next section we explain our model driven approach in more detail.

3. The MODELTALK Concept

The high-level architecture of MODELTALK (Figure 2) is similar to the general architecture of integrated development environments (IDEs). The source code processors (Figure 2B) are replicated to provide similar processors for modeling (Figure 2A). The architecture is implemented in an extensible IDE (Eclipse [8]) to yield an integrative model driven development environment.

3.1 Model Sources

In MODELTALK, model source files are textual and they are managed by the IDE just as other source files. The model sources com-

[2] We use a simplified dialect of MODELTALK concrete syntax.

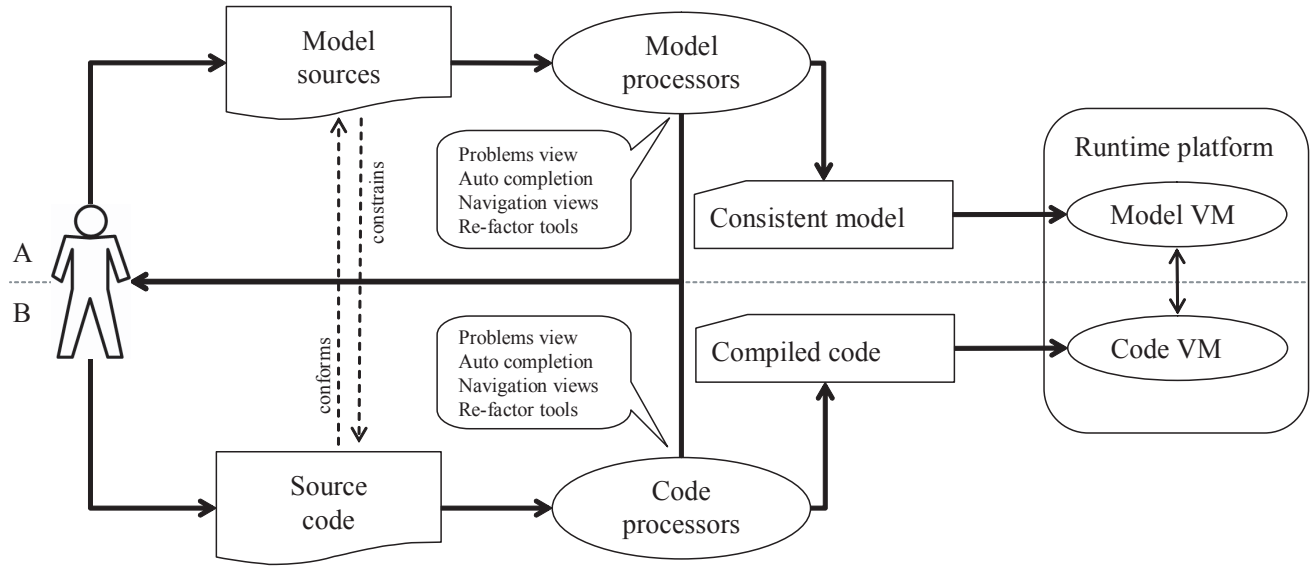


Figure 2. High level architecture of MODELTALK: (A) model processors; (B) code processors

```
<bean id="HTTP_Client" class="Class">
  <properties>
    <property>
      <name>numberOfRetries</name>
      <type>Long</type>
      <description>Number of retries</description>
    </property>
    <property>
      <name>timeout</name>
      <type>Long</type>
      <description>Timeout in seconds</description>
    </property>
    <property>
      <name>URL</name>
      <type>String</type>
      <description>The target URL</description>
    </property>
  </properties>
</bean>
<bean id="RobustHTTP_Client" class="HTTP_Client"
       abstract="true">
  <numberOfRetries>8</timeout>
  <timeout>15</timeout>
</bean>
<bean id="FastHTTP_Client" class="HTTP_Client"
       abstract="true">
  <numberOfRetries>2</timeout>
  <timeout>2</timeout>
</bean>
<bean id="PontisLogoRetriever" class="HTTP_Client"
       parent="FastHTTP_Client">
  <URL>www.pontis.com/logo.bmp</URL>
</bean>
```

Listing 4. Domain model in MODELTALK

prise instances, classes and metaclasses. A domain specific modeling language (DSML) [20] is formed by defining metaclasses and classes. In a typical scenario, the domain expert in the development team defines a DSML. Domain users then define models in this DSML. Since the modeling tools rely on class defined in the model rather than in the code, the modeling activity does not depend on the existence of implementation code.

3.2 Model Compiler

The model compiler is one of the model processors in the MODELTALK framework (Figure 2). It is implemented as an Eclipse builder plug-in. The compiler implements a dependency analysis algorithm to support incremental compilation.

Upon a change to the model, the compiler is invoked to perform cross-model validation. Object graphs in the model are validated against the corresponding model class definitions. This activity is analogous to how a compiler reports syntactical and certain semantical errors. Since MODELTALK is a meta-level system, classes, too, undergo similar validation checks. Cross-checks are necessary because a change in one model element might invalidate other model elements (possibly in other model files).

The model compiler also checks the conformance of the Java sources to the model [18]. When developing the code classes, the tool verifies conformance of the code structure to the model class definitions. Mismatches are reported as errors in the IDE standard problems view.

3.3 Model VM

The model VM is the runtime component of MODELTALK, which is analogous to the JVM. Its primary responsibility is to manage the relationships between model elements and Java elements. This includes object graph instantiation and a reflection API [14].

The model VM implements a dependency injection mechanism. When a client requests a model instance, the Model VM finds the corresponding Java class, instantiates it, and injects the model property values into the Java instance variables. This is applied recursively for injected value of a complex type. The Model VM algo-

rithm for mapping model classes to Java classes permits "holes," i.e., a model class without a Java counterpart. In such a case, the class is called *declarative* and mapped instead to the superclass in the Java model. This adaptability enables to make changes to the model at runtime without needing to also change the Java model.

When a client requests a model class, the Model VM follows the same routine, thus enabling runtime modifications to the model. This is possible because MODELTALK's meta-meta-model itself is implemented in MODELTALK.

3.4 The User Experience in Modeling

The modeling user experience is similar to the user experience in programming. We use a commercial third-party XML editor as our model editor. The model editor provides auto-completion based on XML schema (XSD) generated from the model by the model compiler. Model elements are maintained in multiple source files arranged in folders by XML namespaces.

Model compilation is incremental, providing the user with short response time. Model compilation is done across all model files. Errors are reported to the developer using the standard IDE problems view. The developer can navigate to the erroneous model element by double clicking on the error [1].

The Eclipse plug-in provides numerous views of the model (e.g., type hierarchy) and provides navigation capabilities both between model elements themselves and between the model elements and corresponding Java elements. In addition, the plug-in provides refactoring facilities (e.g., rename) that propagate the changes to the Java source as well. Model source files are managed in a central repository (CVS) as other source files.

4. Meta-modeling with MODELTALK

In this section we illustrate the domain specific modeling capabilities of MODELTALK by extending the HTTP client example presented in Section 2. Suppose we would like to cache data in order to reduce network traffic and to improve the overall response time. Lets assume the application uses `HTTP_Client` to retrieve different kinds of data: pictures, news, stock quotes, etc. Obviously, various kinds of data require different caching policies. For example, pictures can be cached for longer periods than news, while stock quotes shouldn't be cached at all.

4.1 Declarative Classes

Implementing the caching code in Java in each of these classes would require the expertise of a Java developer. Instead, we can define a metaclass `MetaCache` with a `cache` property of type `CacheManager` (Figure 3 and Listing 5). The `CacheManager` provides cache management services at runtime. The methods `getFromCache` and `putInCache` are defined in the `CacheManager` class in Java. For brevity, the `CacheManager` and `StandardCache` classes are not shown in the listing.

We now make the `HTTP_Client` class an instance of `MetaCache`. The `sendReceive` method in `HTTP_Client` may then use the `MetaCache` metaclass to access the cache (Listing 6). We can further define specific HTTP client classes, `PictureRetriever`, `NewsRetriever`, and `StockQuoteRetriever` (Figure 4 and Listing 7), by subclassing `HTTP_Client`. Note that `PictureRetriever`, `NewsRetriever`, and `StockQuoteRetriever` are *declarative* (without a counterpart in Java).

4.2 Customizing a Metaclass

Next, we enhance the example to demonstrate how architectural definitions are enforced by MODELTALK. Suppose our application needs to display bank account balances that are also retrieved using HTTP. Since bank account information is private, its confidentiality should be kept. We therefore have to

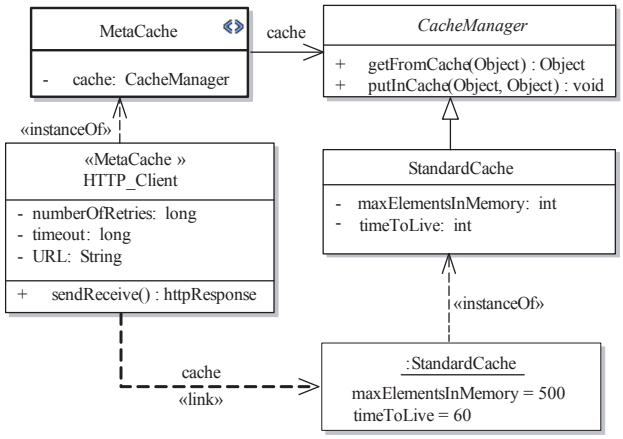


Figure 3. Domain model with a custom metaclass

```
<bean id="MetaCache" class="Class" parent="Class">
  <properties>
    <property>
      <name>cache</name>
      <type>CacheManager</type>
      <description>Caches the result</description>
      </property>
  </properties>
</bean>
<bean id="HTTP_Client" class="MetaCache">
  <cache class="StandardCache">
    <timeToLive>60</timeToLive>
    <maxElementsInMemory>500</maxElementsInMemory>
  </cache>
  <properties>
  ...
  </properties>
</bean>
```

Listing 5. MODELTALK sources with a custom metaclass

```
public HttpResponse sendRecieve() {
  MetaCache myMetaclass =
      (MetaCache)Kernel.instance().getClass(this);
  HttpResponse result =
      myMetaclass.getCache().getFromCache(getURL());
  if (result == null) {
    // do the business logic using timeout & numberOfRetries
    myMetaclass.getCache().putInCache(getURL(), result);
  }
  return result;
}
```

Listing 6. Accessing a metaclass in Java

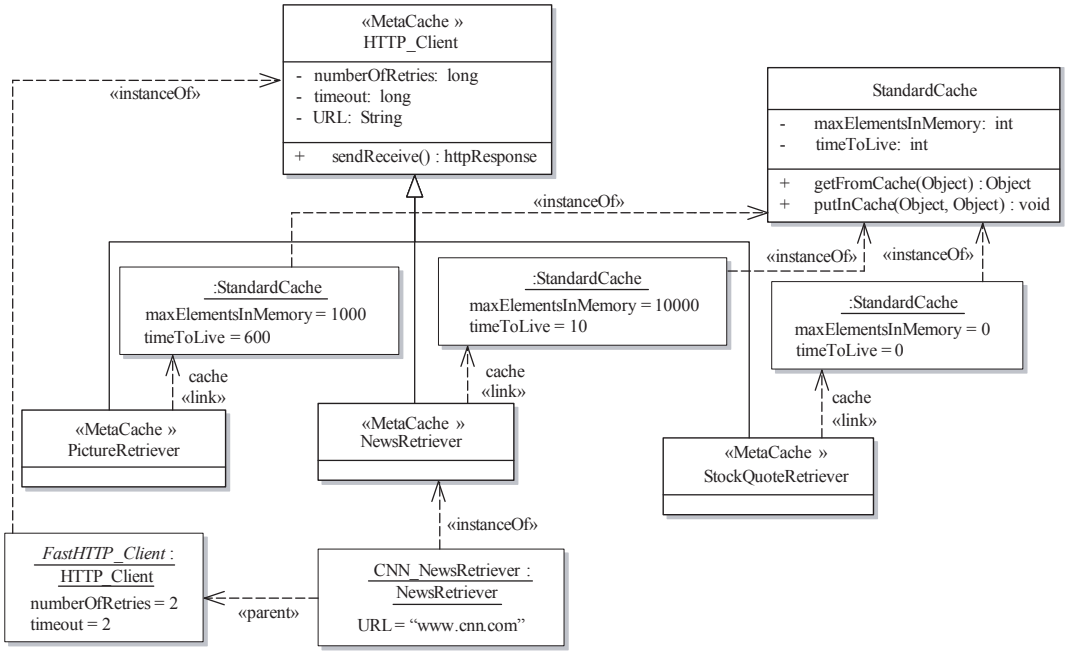


Figure 4. Expanding the domain model with different types of resources

```
<bean id="PictureRetriever" class="MetaCache" parent="HTTP_Client" declarative="true">
  <cache class="StandardCache">
    <timeToLive>600</timeToLive>
    <maxElementsInMemory>1000</maxElementsInMemory>
  </cache>
</bean>
<bean id="NewsRetriever" class="MetaCache" parent="HTTP_Client" declarative="true">
  <cache class="StandardCache">
    <timeToLive>10</timeToLive>
    <maxElementsInMemory>10000</maxElementsInMemory>
  </cache>
</bean>
<bean id="StockQuoteRetriever" class="MetaCache" parent="HTTP_Client" declarative="true">
  <cache class="StandardCache">
    <timeToLive>0</timeToLive>
    <maxElementsInMemory>0</maxElementsInMemory>
  </cache>
</bean>
<bean id="CNN_NewsRetriever" class="NewsRetriever" parent="FastHTTP_Client">
  <URL>www.cnn.com</URL>
</bean>
```

Listing 7. The resource retrieving model in MODELTALK

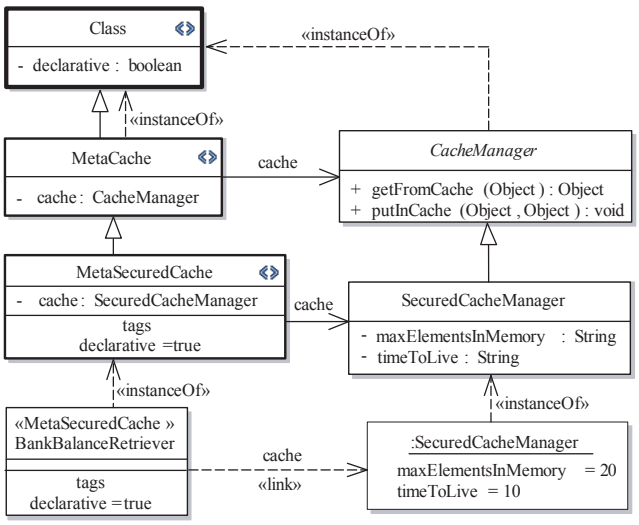


Figure 5. Expanding the domain model with secured cache

```
<bean id="MetaSecuredCache" class="Class"
      parent="MetaCache" declarative="true">
  <properties>
    <property>
      <name>cache</name>
      <type>SecuredCacheManager</type>
      <description>Provides secured caching
          capabilities</description>
    </property>
  </properties>
</bean>
<bean id="BankBalanceRetriever"
      class="MetaSecuredCache"
      parent="HTTP_Client" declarative="true">
  <cache class="SecuredCacheManager">
    <timeToLive>10</timeToLive>
    <maxElementsInMemory>20</maxElementsInMemory>
  </cache>
</bean>
```

Listing 8. The secured cache model in MODELTALK

make sure that only a secured cache manager is used in such cases. To achieve this we define `MetaSecuredCache` that extends `MetaCache` and overrides the type of the *cache* property to `SecuredCacheManager`, which is a subclass of `CacheManager` (Figure 5 and Listing 8). `BankBalanceRetriever` is defined as instance of `MetaSecuredCache` and is therefore required to supply an instance of a `SecuredCacheManager` for its cache property; otherwise, a model compilation error will be issued. For the sake of brevity, we do not show the corresponding changes in the Java code.

Custom metaclasses [5, 6] define class level properties, which constrain the domain classes using domain terminology. Unlike class level members in Java (static members), metaclass properties allow subclasses to have different values for the metaclass properties.

The model contains objects, classes and metaclasses in a single type system. The same injection mechanism that works on objects is applied to classes as objects of their metaclasses. Since Java does not support metaclass extensibility, the metaclasses in the model are mapped to regular Java classes and the model VM manages the *instance-of* relation in the runtime system.

5. Assessment

The MODELTALK development platform has been used at Pontis by a team of over 20 developers for the last two years. Numerous customer projects were developed, delivered and deployed successfully, satisfying requirements for hundreds of transactions per second. In this section we describe our subjective observations from using MODELTALK in a commercial product development environment.

Since the inception of the platform, we have been continuously running code measurements in order to indicate the platform's level of adoption within the R&D organization. Currently, the model contains approximately 4800 classes, of which 200 are metaclasses. There are tens of thousands of instances and 275K lines-of-code in XML. The average Depth of Inheritance (DIT) of model classes is 4.75. The XML schema of the application model (generated XSD) is 200K lines-of-code. 90% of the application Java source code is governed by the model (i.e., the classes are defined in the model). 82% of the source lines-of-code in customization projects are declarative.

5.1 Developer Perspective

Developers give very positive feedback, mostly concerning the dramatic improvement in cycle times. An incremental model change takes no more than a few seconds on large models, compared to minutes, at best, in generative MDD (e.g., [23, p. 256] and [23, p. 261]).

Users appreciate the Java-like usability of MODELTALK and the fact that modeling and programming are done in a single, integrated environment. Specifically, the developers mention the ability to work on "broken models." For example, when changing a property name in a class, instances of the class become invalid. The MODELTALK environment allows the developer to continue modeling although part of the model is temporarily in an inconsistent state.

Users get accustomed to formal modeling very quickly and rely on the model compiler to enforce architectural constraints. Users complained about the tedious, manual work in writing the structural part of the Java code, especially getters/setters. To address this, we extended MODELTALK with some code generation capabilities, which is outside the scope of this paper. The generation of getters and setters provides Java type-safety, but is strictly optional, because the model is fully reflective.

Users also complain about the lack of diagramming capabilities and interoperability with UML modeling tools. This is a topic we plan to address in future work.

5.2 Organization Perspective

An organization considering adopting a similar approach should take into account a substantial initial investment for building the infrastructure and tools. In our case, the investment was more than 10 man years. In addition, the ongoing maintenance of the development environment must be considered. Another barrier is the inherent complexity of such an approach. Most developers are not familiar with meta-modeling and need extensive training. Moreover, application implementation tends to be highly abstract and generic, which requires highly talented individuals.

However, once such a development environment is in place, there are tangible benefits for the organization. First and foremost,

all the advantages of MDD and DSML apply [23, 22]. Especially, time-to-market and the cost of producing customized products drop significantly. Second, the ability to deploy a compiled model directly to a running system (when changes in Java are not required) creates a much shorter delivery route.

6. Conclusion and Future Work

Software solutions in the telecommunications industry typically require massive customization. In order to reduce the cost and time-to-market of creating customized products, we developed MODELTALK, a domain specific model driven framework, and used the framework in product development.

An early implementation of MODELTALK was based on a generative architecture centric MDSD approach [23]. The advantages of the model centric approach were evident. However, when the model evolved to thousands of classes, developers started to complain about the long development cycles (several minutes for each incremental change). This stemmed from the large amounts of generated Java code that had to undergo compilation, packaging and deployment to the J2EE application server.

In this paper we presented the new version of MODELTALK, which is based on an executable model, dependency injection, and meta-modeling; and complemented by a model compiler and tooling. Together these provide an enhanced user experience for the modeling process, similar to the programming user experience in modern IDEs.

The meta-level capabilities of MODELTALK are used by developers to create custom types of classes, fields and methods resulting in a domain specific modeling language. There is also support for resolving crosscutting concerns at the model level [3, 2, 4, 7, 13].

We are currently working on enhancing MODELTALK with runtime adaptability, i.e., the ability of non-programmers to change the model of a production system [9, 16]. The interpretive nature of the MODELTALK platform provides a sound basis for achieving this by combining an interpretive approach with metaclass extensibility [5].

Acknowledgments

We thank Zvi Ravia and Shai Koenig for their valuable comments. We thank our colleagues in Pontis and partners in developing the MODELTALK platform: Shachar Segev, Moshe Moses and Assaf Pinhasi. Pontis Ltd. is a developer of online marketing automation solutions for the telecommunications market.

References

[1] J. Bezivin and F. Jouault. Using ATL for checking models. In *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT)*, 2005.

[2] N. Bouraqadi. Efficient support for mixin-based inheritance using metaclasses. In *GPCE'03 Workshop on Reflectively Extensible Programming Languages and Systems*, 2003.

[3] N. Bouraqadi. Safe metaclass composition using mixin-based inheritance. *Journal of Computer Languages and Structures*, 30(1-2):49–61, 2004.

[4] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of ECOOP/Object-Oriented Programming Systems, Languages, and Applications*, pages 303–311, Ottawa, Canada, 1990. ACM SIGPLAN Notices 25(10) Oct. 1990.

[5] J. P. Briot and P. Cointe. Programming with explicit metaclasses in Smalltalk-80. In *Proceedings of the 4th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'89)*, pages 419–431, New Orleans, Louisiana, 1989. ACM SIGPLAN Notices 24(10) Oct. 1989.

[6] S. Danforth and I. R. Forman. Reflections on metaclass programming in SOM. In *Proceedings of the 9th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)*, pages 440–452, Portland, Oregon, USA, 1994. ACM SIGPLAN Notices 29(10) Oct. 1994.

[7] S. Ducasse, N. Schärli, and R. Wuyts. Uniform and safe metaclass composition. *Journal of Computer Languages and Structures*, 31(3-4):143–164, 2005.

[8] Eclipse development platform. `http://www.eclipse.org/`.

[9] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, 2006.

[10] M. Fowler. Inversion of control containers and the dependency injection pattern. `http://martinfowler.com/articles/injection.html`.

[11] M. Fowler. Language workbenches: The killer-app for domain specific languages? `http://martinfowler.com/articles/languageWorkbench.html`.

[12] Business process execution language for web services. `http://www.ibm.com/developerworks/library/specification/ws-bpel/`.

[13] S. Kojarski and D. H. Lorenz. Identifying feature interactions in aspect-oriented frameworks. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, Minneapolis, MN, May 20-26 2007. IEEE Computer Society.

[14] D. H. Lorenz and J. Vlissides. Pluggable reflection: Decoupling meta-interface and implementation. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 3–13, Portland, Oregon, May 1-10 2003. IEEE Computer Society.

[15] D. R. Prasanna. *Dependency Injection*. Manning, 2008.

[16] R. Razavi, N. Bouraqadi, J. W. Yoder, J.-F. Perrot, and R. Johnson. Language support for adaptive object-models using metaclasses. *Computer Languages, Systems and Structures*, 31(3-4):188–218, October-December 2005.

[17] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe. The architecture of a UML virtual machine. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*, pages 327–341, Tampa Bay, Florida, Oct. 14-18 2001. ACM SIGPLAN Notices 36(11) Nov. 2001.

[18] E. Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, 2003.

[19] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[20] B. Selic. A systematic approach to domain-specific language design using uml. In *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*. IEEE Computer Society, 2007.

[21] Spring framework. `http://www.springframework.org/`.

[22] A. Uhl. Model-driven development in the enterprise. *IEEE Software*, 24(1):46–49, 2008.

[23] M. Völter and T. Stahl. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

[24] J. W. Yoder and R. E. Johnson. The adaptive object-model architectural style. In *WICSA3 '02: Proceeding of The 3rd IEEE/IFIP Conference on Software Architecture*, pages 3–27, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

SMML: Software Measurement Modeling Language

Beatriz Mora
Alarcos Research Group,
Department of Computer Science,
University of Castilla-La Mancha
beatriz.mora@uclm.es

Felix García
Alarcos Research Group,
Department of Computer Science,
University of Castilla-La Mancha
felix.garcia@uclm.es

Francisco Ruiz
Alarcos Research Group,
Department of Computer Science,
University of Castilla-La Mancha
francisco.ruizg@uclm.es

Mario Piattini
Alarcos Research Group,
Department of Computer Science,
University of Castilla-La Mancha
mario.piattini@uclm.es

Abstract

Domain Specific Languages (DSLs) and Software Measurement are at present increasingly important in Software Engineering research.

Domain Specific Languages (DSLs) and Software Measurement are at present increasingly important in Software Engineering research

They have, in fact, become important aspects of the software industry. Domain languages facilitate the software development process in a specific domain, and measurement can help to address certain critical issues in software development and maintenance by facilitating the making of decisions. This work presents a language which allows users to define software measurement models based on the Software Measurement Ontology. Syntactically and semantically correct models in this language conform to a specific measurement metamodel, which is aligned with the aforementioned ontology.

Keywords SMML, DSL, Software Measurement

1. Introduction

Software Measurement has become a fundamental aspect of Software Engineering [1]. Measurement is proving to be highly effective in, among other things, the construction of high quality prediction systems for large-scale data base projects [2]; in the understanding and improvement of software development and maintenance projects [3]; in the evaluation and guarantee of system quality (by highlighting problematic areas) [4]; and in the determination of better work practices with the goal of assisting users and investigators in their work [4]. Moreover, software measures assist in the evaluation and institutionalization of Software Process Improvement in those organizations which develop them. Software Measurement is, in fact, a key element in initiatives such as SW-CMM (Capability Maturity Model for Software), ISO/IEC 15504 (SPICE, Software Process Improvement and Capability dEtermination) and CMMI (Capability Maturity Model Integration) [5]. The ISO/IEC 90003:2004 standard [6] also highlights the importance of measurement in managing and guaranteeing quality. Various methods and standards with which to carry out measurements in a precise and systematic manner exist, of which the most representative are:

- **Goal Question Metric (GQM):** The basic principle of GQM is that the carrying out of the measurement must always be oriented towards an objective. GQM defines an objective, refines that objective into questions and defines measures which attempt to answer those questions.
- **Practical Software and Systems Measurement (PSM):** The PSM methodology [7] is based upon the experience obtained from organizations through which the best manner in which to implement a software measurement programme with guarantees of success is discovered.
- **IEEE 1992 (Methodology for Software Quality Measures):** according to the IEEE 1992 standard, software quality can be considered as the extent to which the software possesses a clearly defined and desirable combination of quality attributes.
- **ISO/IEC 15939:** this international standard [8] identifies the activities and tasks which are necessary to successfully identify, define, select, apply and improve software measurement within a general project or within a business measurement structure.

The availability of a language which allows users to represent those elements which must be taken into account in the measurement processes might, therefore, be important in decision making and in process improvement.

It is thus of interest to consider the use of Domain Specific Languages (DSLs). DSLs appear in the context of Domain Specific Modeling (DSM). Domain-Specific Modeling raises the level of abstraction beyond programming by specifying the solution with the direct use of domain concepts. The final products are generated from these high-level specifications. This automation is possible because both the language and generators need to fit the requirements of only one company and domain. Industrial experiences of DSM consistently show it to be 5-10 times more productive than current software development practices [9], including current UML-based implementations of MDA. DSM does to code what compilers did to assembly language. Besides this vision, more investigation is needed in order to advance the acceptance and viability of DSM. Selection of a domain is a first step towards development of domain-specific languages which implies trade offs between more general applicability of the DSL and more specificity [10]. In other words, a trade off between the focus and size of the language is needed. A language which represents a larger domain can be weakly specialized to any particular aspect of the domain. On the contrary, a language which represents a small domain may have a limited number of target users [11].

These aspects constitute the main interest of this paper, whose objective is to propose the Software Modeling Measurement Language (SMML) which will permit software measurement models to be created in a simple and intuitive manner. This language has been done by using the Software Measurement Metamodel (SMM) [12] (for greater detail see [13]) as the Domain Definition Metamodel (DDMM). This language belongs to the Software Measurement Framework (SMF) presented in [14] and also dis-

cussed in Section 3 of this paper. SMF allows stakeholders to obtain generic measurement through transformations by using two initial models as a starting point: that of software measurement and that domain. The task of the SMML is to facilitate the definition of software measurement models, which is the starting point of generic software measurement processes.

The remainder of the paper is organized as follows. Section 2 provides an overview of related works and Section 3 briefly describes SMF. In Section 4 SMML is explained, including the definition of the abstract syntax, concrete syntax and semantics. Section 5 illustrates the use of SMML in the context of a case study. Finally, conclusions and future work are outlined in Section 6.

2. Related Work

There are numerous works related to the development of DSLs. On the one hand we can find publications which present methodologies, proposals, tools and patterns with which to facilitate the development of DSLs [15-20].

In [15] is proposed that the next step towards developing a technology for software manufacturing is the development of DSLs.

In order to aid the DSL developer, [16] identifies patterns in the decision, analysis, design, and implementation phases of DSL development. These patterns improve and extend earlier work on DSL design patterns. They also discuss domain analysis tools and language development systems which may help to speed up DSL development.

In [17] is presented a partial requirements analysis for DSLs in general, focusing on relevant stakeholders, the system boundary (i.e., where DSLs end and general purpose languages start), and a core set of requirements which are relevant for any DSL. They then discuss open questions, focusing particularly upon requirements refinement, in which more specific domain information needs to be used. Their discussion is intended to be generic: they do not distinguish between domain-specific modeling and programming languages (except where noted). They therefore refer to descriptions as the construct produced by using a DSL. Specific instances of descriptions may be models or programmes.

A study of the literature available on the topic of DSLs as used for the construction and maintenance of software systems is presented in [19]. The authors list a selection of 75 key publications in the area, and provide a summary for each of the papers. Moreover, they discuss terminology, risks and benefits, example domain-specific languages, design methodologies, and implementation techniques.

Numerous works presenting DSL exist: ATL (ATLAS Transformation Language) [21], a QVT-like model transformation language [22] and its execution environment which is based on the Eclipse framework; KM3 (Kernel MetaMetaModel) [23] which is a DSL for describing metamodels; etc.

With regard to DSLs for Software Measurement, Guerra et al. [24] present a framework for the creation of domain specific visual languages (DSVL). In this work a language called SLAMMER was developed as a case study. This language is part of the suite of model management tools that Guerra et al. have defined using graph grammars and graph transformations, in which the evaluation and measurement of software artefacts is an essential element. The goal is to facilitate the task of defining measurements and redesigns for any DSVL.

The Software Metrics Meta-Model [25] developed by the OMG also exists. The Software Metrics Meta-Model, promotes a common interchange format which allows interoperability between existing modernization tools, services and their respective models. This common interchange format can be applied equally well to development and maintenance tools, services and models. In spite of the existence of this Metamodel, we have opted to define our own language owing to the fact that the Software Measurement Ontology [26] exits. This ontology permits us to establish and clarify the elements (concepts and relationships) involved in the software measurement domain. We have, therefore, based the definition of SMML on this ontology. We have verified that the use of this ontology provides important advantages, particularly given the importance of the solid conceptual base that the problem domain (ontology) provides with which to be able to tackle the solution domain (metamodel). The ontologies are, moreover, potentially useful when developing DSLs during the analysis phase in which knowkedge capture and knowledge representation are the key elements [27].

3. Software Measurement Framework

The Software Measurement Framework (SMF) (for greater detail see [14]) permits us to measure any type of software entity. In this framework, any software entity in any domain can be measured with a common Software Measurement metamodel and QVT transformations. SMF has three fundamental elements: conceptual architecture, technological aspects and method. These elements have all been adapted to the MDE paradigm and to MDA technology, taking advantage of their benefits within the field of software measurement. The Software Measurement Framework (SMF) is the evolution of the FMESP [28], but is adapted to the MDE paradigm and uses MDA technology.

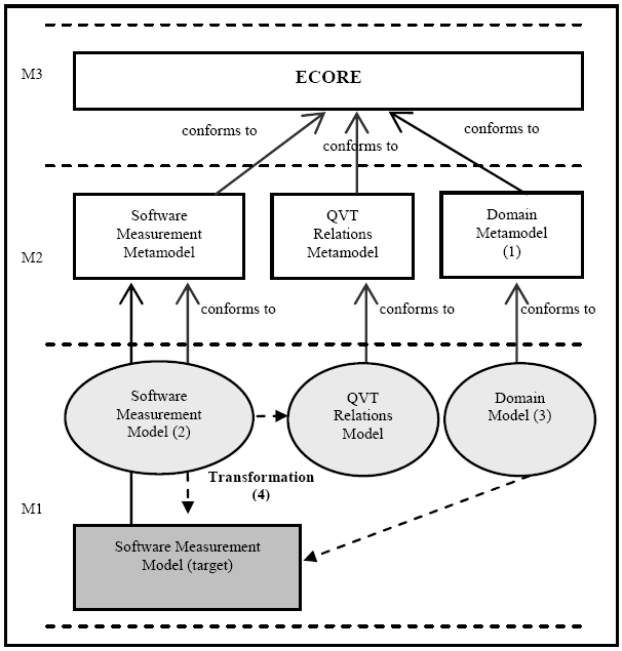


Figure 1. Elements of the SMF.

In Figure 1 the necessary elements for the adaptation of FMESP to MDA are presented according to MOF levels.

4. Software Measurement Modeling Language (SMML)

SMML is a language which permits software measurement models to be built in a simple and intuitive manner. The SMML development requires both domain knowledge and language development expertise [16].
Feilkas [29] cites the tasks which must be carried out to make a DSL usable: Definition of an abstract syntax, Definition of a concrete syntax and Definition of semantics. The following subsection describes how these stages have been used to develop the Software Measurement Modeling Language (SMML) [30].

4.1 Definition of an abstract syntax (Domain definition metamodel)

One of the defining entities of a DSL is a Domain Definition MetaModel (DDMM) [30]. This introduces the basic entities of the domain and their relationships. This base ontology plays a central role in the definition of the DSL. Such a DDMM plays the role of the abstract syntax for a DSL.

In order to develop SMML, a Domain Definition Metamodel is therefore necessary. The Software Measurement Metamodel (SMM) exists, which is derived from the Software Measurement Ontology (SMO). This metamodel is the Domain Definition Metamodel used to define the abstract syntax of SMML.

The Software Measurement Metamodel includes the packages which are alignments with the sub-ontologies of SMO (Basic, Characterization and Objectives, Measures Software, Measurement Approaches and Measurement Action). However, for the development of the Language, all the packages are of interest, with the exception of Measurement Action. This has been excluded as it contains the elements which are relative to measurement but not to the problem domain. Figure 2 shows the structure of the packages upon which the SMML language is based.

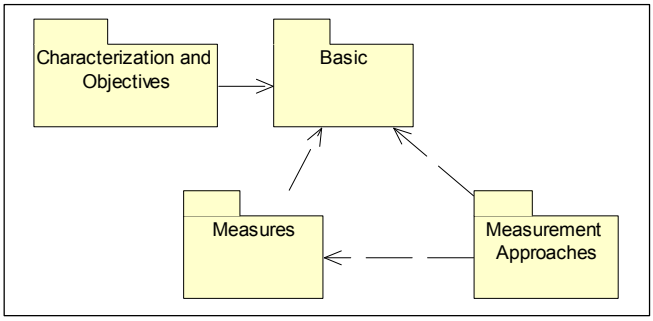


Figure 2. Structure of the packages in the Software Measurement Metamodel.

As shown in Figure 2, the metamodel is made up of a basic package which represents the general characteristics of the basic constructors of the measurement models, and three other packages (Characterization and Objectives, Measurement Approaches and Measures), in accordance with the three sub-ontologies of the SMO. The conceptualization established in the Software Measurement Ontology has been taken into account in the construction of this metamodel, but the specific constructors have been added from the point of view of implementation.

All of the elements identified in the ontology (Measure, Information need, Measurable concept, etc.) are potential elements of the Software Measurement Metamodel on which the SMML language is based. On the other hand, the relationships which exist in the ontology do not correspond with the relationships which are necessary for the language. All of the Measurement Metamodel packages maintain the original definition of [12] with the exception of the basic package, which has had to be adapted to represent the measurement relationships in SMML.

In [13] is given a detailed description of the relationships of the Software Measurement Metamodel which correspond with the relationships in the SMO ontology. As [13] shows, all the types of relationships which are identified in the ontology, and which have been defined for the metamodel, have been studied. The elements involved (a source and a target) are indicated for each relationship. In total, 4 types of Measurement Associations have been identified: association, nonnavigable association, aggregation and dependency. These relationships have been defined in the Basic package.

In the following table a selection of relationships in the "software measurement characterization and objectives" package are shown. Note that for each relationships in the SMO we have related a new Measurement Association (see Table 1).

Table 1. A selection of the SMML elements and icons

Relationships	Description	Source
Includes	An *entity class* may include several other *entity classes*. An *entity class* may be included in several other *entity classes,*	UML Aggregation
Defined for	A *quality model* is defined for a certain *entity class*. An *entity class* may have several *quality models* associated	UML Dependency
Relates	A *Measurable concept* relates one or more *attributes.* An *Attribute* is related with one or more *measurable concepts*.	UML Association
Has	An *entity class* has one or more *attributes*. An *attribute* can only belong to one *entity class*.	UML nonnavigable Association

We shall now describe the packages of which the Software Measurement Metamodel is made up (for greater detail see [13]):

- **Basic Package:** this basic package has been defined in order to identify and to establish the general features of the constructor necessary to define measurement model. With regard to the Software Measurement Metamodel defined in [12], 4 types of *Measurement Association* have been added: association, nonnavigable association, aggregation and dependency. Figure 3 shows the UML diagram which displays the structure of this package.

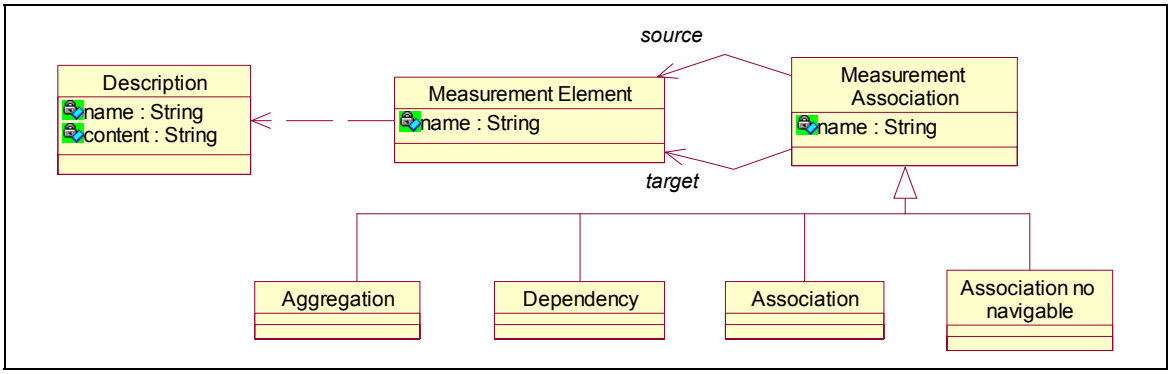


Figure 3. Basic package.

As can be observed in Figure 3, the general element from which measurement models are constructed is the "Measurement Element" constructor, and the general element from which the relationships of the models are constructed is the "Measurement Association" constructor. A measurement element has a name and can be described through elements of the "Description" type, which give additional information about the measurement elements, and this facilitates a better understanding of the measurement models developed. The measurement element is used as a starting point from which to specialize the measure's fundamental constructors, obtained from the Software Measurement Ontology concepts. A Measurement Element relates two measurement elements, a source element and a target element. The Measurement Association is used to specialize the relationship constructors defined for the metamodel: Association, Nonnavigable association, Aggregation and Dependency.

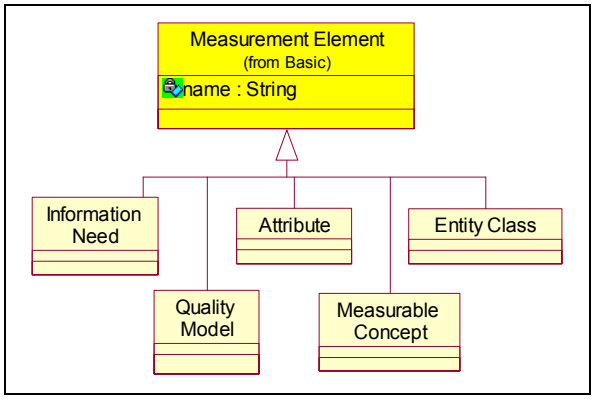


Figure 4. Characterization and objectives Package.

- **Characterization and objectives Package:** this package includes the constructors required to establish the scope and objectives of the software measurement process. Figure 4 shows the UML diagram which displays the structure of this package.
- **Software Measures Package:** this package includes the constructors needed to establish and to clarify the key elements in the definition of a software measure. Figure 5 shows the UML diagram which displays the structure of this package.

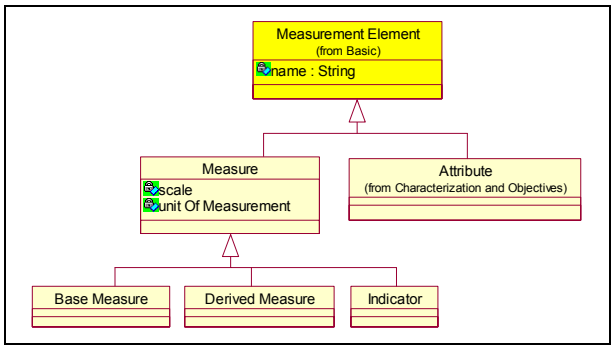


Figure 5. Software Measures Package.

- **Measurement Approaches Package:** this package includes the constructors needed to generalize the different 'approaches' used by the three kinds of measures to obtain their respective measurement results. Figure 6 shows the UML diagram which displays the structure of this package.

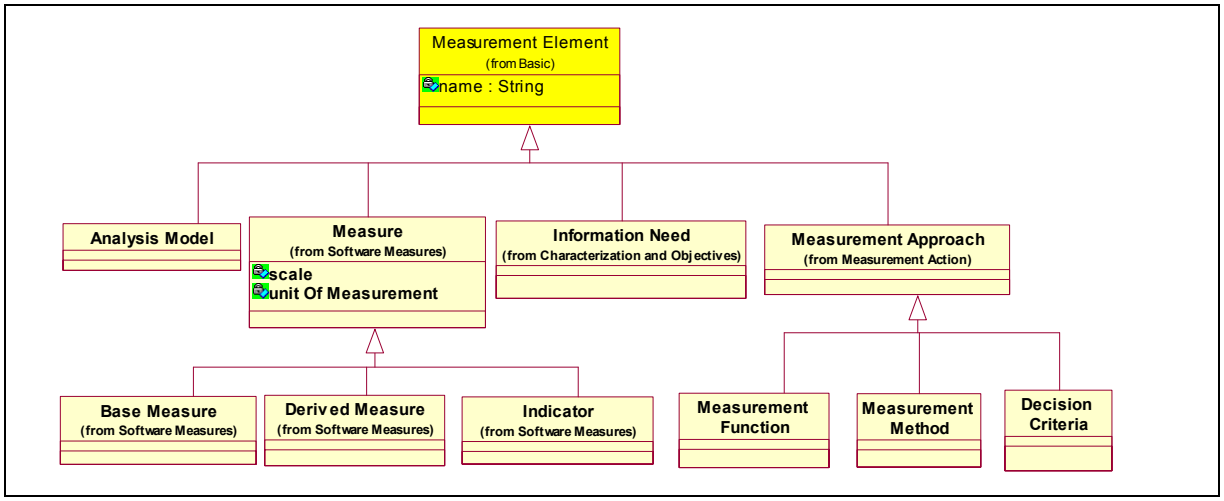


Figure 6. Measurement Approaches Package.

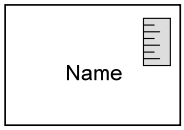
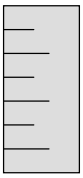
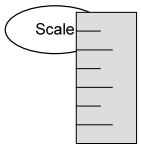
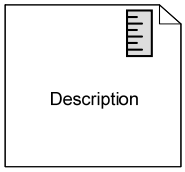
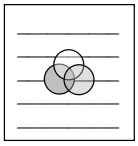
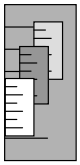
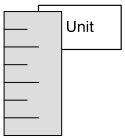
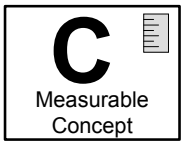
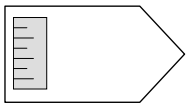
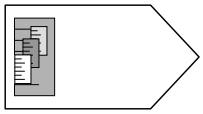
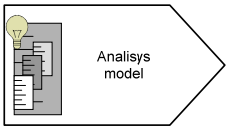
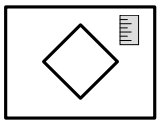
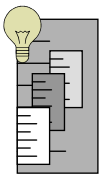
4.2 Definition of a concrete syntax

In order to make the language usable, a concrete syntax must be defined. All of the elements are defined in the basic package (see Figure 3).

Each of these elements of the language must be associated with a graphical icon which represents the element of the abstract model. Each language element and relationship has been associated with a representative icon in the SMML. Icons which are familiar to software engineers have been used in order to facilitate its use. For example, the Description element is very similar to the UML note element, the difference being that the former includes a ruler icon while the latter does not (as a symbol of measurement) in its top right-hand corner. In a similar manner, the Entity element is taken from the Entity Class in an E/R Diagram.

Table 2 shows a selection of the language elements. For further information, see [13]:

Table 2. A selection of the SMML elements and icons

Information need	Entity	Base Measure	Scale	Description
 Information need	 Name		 Scale	 Description
Quality Model	**Attribute**	**Derived Measure**	**Unit**	**Measurable Concept**
 Quality Model	 Attribute		 Unit	 C Measurable Concept
Measurement Method	**Measurement Function**	**Analysis model**	**Decision Criteria**	**Indicator**
		 Analisys model		

4.3 Definition of semantics

The most important aspect of language specification is possibly the definition of its semantics. An informal description of the language must be given in a natural language which describes its domain. The semantics of the language have been defined by using OCL constraints on the metamodel. These constraints define the cardinality and the elements involved in the associations. These constraints are considered too as being part of the abstract syntax because they are part of the metamodel. An example of OCL Constraints relating to Measures is shown in Table 3:

Table 3. A selection of SMML OCL Constraints.

OCL Constraint
Element: Nonnavigable Association
`self.source.oclIsTypeOf(`***EntityClass***`) and self.target.oclIsTypeOf(`***Attribute***`)`
Element: Association
`self.source.oclIsTypeOf(`***MeasurableConcept***`) and self.target.oclIsTypeOf(`***Attribute***`)` **or** `self.source.oclIsTypeOf(`***DerivedMeasure***`) and self.target.oclIsTypeOf(`***MeasurementFunction***`)` **or** `self.source.oclIsTypeOf(`***BaseMeasure***`) and self.target.oclIsTypeOf(`***MeasurementMethod***`)` **or** `self.source.oclIsTypeOf(`***Indicator***`) and self.target.oclIsTypeOf(`***AnalysisModel***`)`
Element: Agregation
`self.source.oclIsTypeOf(`**EntityClass**`) and self.target.oclIsTypeOf(`**EntityClass**`)`
Element: Dependency
`(self.source.oclIsTypeOf(`**QualityModel**`) and self.target.oclIsTypeOf(`**EntityClass**`)) or (self.source.oclIsTypeOf(`**QualityModel**`) and self.target.oclIsTypeOf(`**MeasurableConcept**`)) or (self.source.oclIsTypeOf(`**MeasurableConcept**`) and self.target.oclIsTypeOf(`**InformationNeed**`)) or (self.source.oclIsTypeOf(`**AnalysisModel**`) and self.target.oclIsTypeOf(`**DecisionCriteria**`)) or (self.source.oclIsTypeOf(`**Indicator**`) and self.target.oclIsTypeOf(`**InformationNeed**`)) or (self.source.oclIsTypeOf(`**Measure**`) and self.target.oclIsTypeOf(`**Attribute**`))`

As can be observed, the preceding table (Table 3) contains the OCL constraints which verify whether the Measurement Elements involved in each Measurement Association (source and target) are correct.

5. Case Study

To illustrate the benefits of the SMML, consider the following two case studies: the development and maintenance of database applications in a software company and the definition of a Data Quality Model for Web Portals.

The first case of study is detailed in [31]. This paper presents the results and lessons learned in the application of the Framework for the Modeling and Measurement of Software Processes (FMESP) [28] in a software company dedicated to the development and maintenance of software for information systems.

All the information concerning the problem is defined in each Software Measurement Package: Characterization and Objectives, Software Measures and Measurement Approaches. This case will only show the modeling of the Characterization and Objectives package.

In this example, we wish to illustrate how a measurement model would be represented with SMML. Figure 7 shows all the information that is needed to represent the Characterization and Objectives Instance. The Measurement Elements used are: Information Need, Quality Model, Measurable Concept, and Attribute. This model has been defined by using diagrams of UML objects.

We shall, furthermore, present how the same example would be defined with SMML (Figure 8).

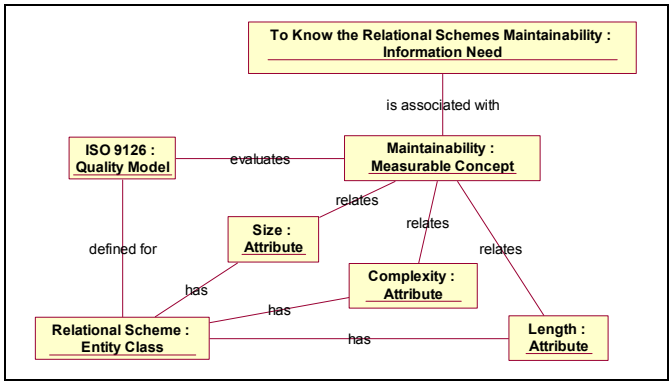


Figure 7. Characterization and Objectives Instance with UML.

As will be observed from the following figure, the representation is easier and more intuitive with the SMML language. Moreover, during the measurement model definition, no issues were found in the constructors metamodel, and no lacks were detected in the Language.

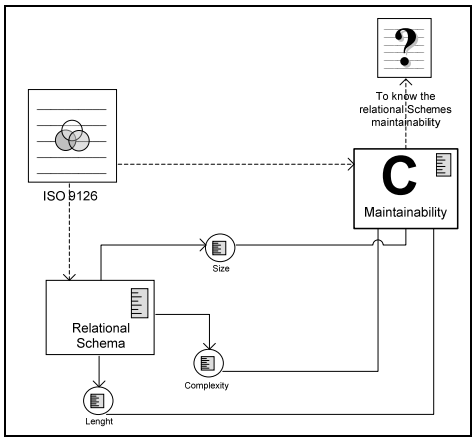


Figure 8. Characterization and Objectives Instance with SMML.

The second case study is shown in [32]. This paper shows how the SMO can be instantiated to define a Data Quality Model for Web Portals, and can also be used to define a DSL for measuring software entities.

Figure 9 shows all the information that is needed to represent the Measurement Model of PDQM. The Measurement Elements used are: *Information Need*, *Quality Model*, *Measurable Concept*, *Attribute, Base Measure, Derived Measure, Indicator, Measurement Method, measurement Function* and *Analysis Model*.

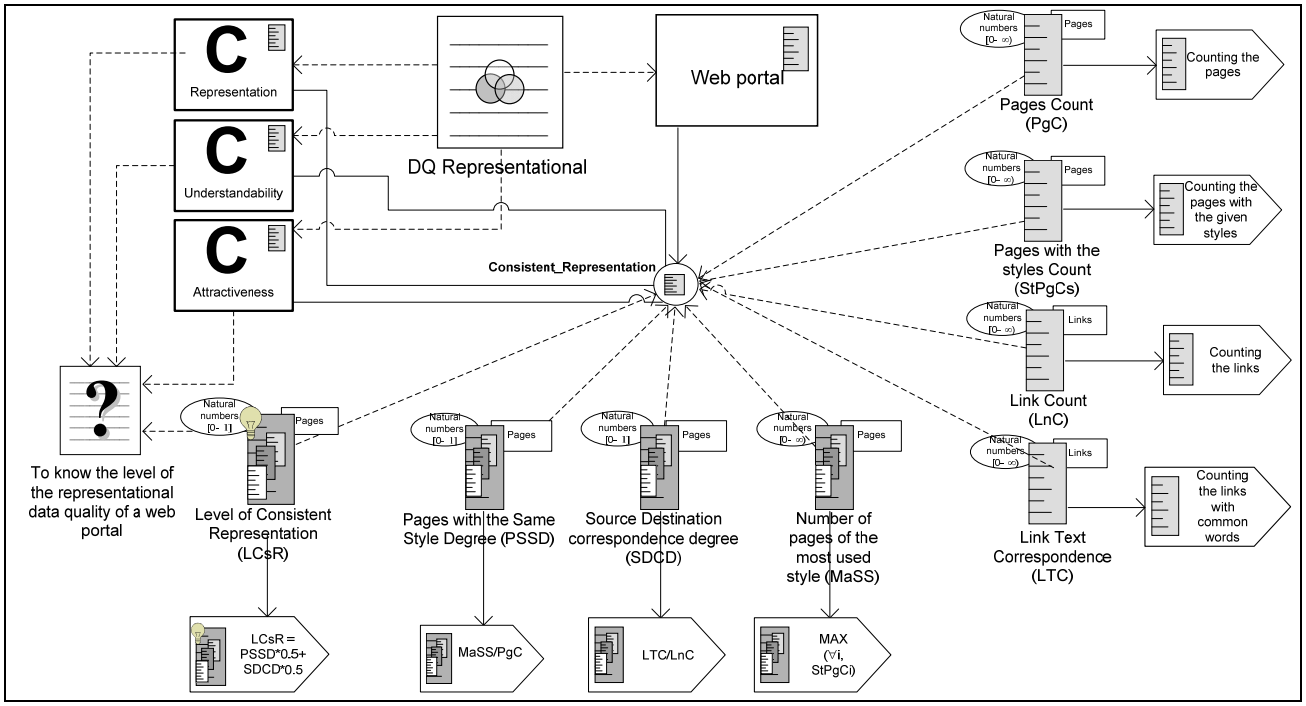


Figure 9. Measurement Model of PDQM represented with SMML.

In this case study, in spite of having to define numerous Measurement Elements, the representation continues to be easy and intuitive. What is more, it is easier to identify Measurement Elements by using this model than by using another General Purpose Language such as UML

With regard to expected requirements [17], we shall now show the requirements which are valid in our Language:

- **Conform:** the language constructs correspond to important domain concepts.
- **Orthogonal:** Each language construct is used to represent exactly one distinct concept (Attribute, Base Measure, etc.) in the domain.
- **Supportable:** The SMML language is supported by tools such as MS/DSL Tools or GMF [33].
- **Simple:** the DSL is simple in order to express the domain concepts and to support its users.
- **Usable:** DSL constructs are expressive and easy to understand.

6. Conclusions and Future Work

SMML permits software measurement models to be defined in a manner which is easy and intuitive for the user. The set of icons which form a part of the language have been selected in order for them to be as familiar as possible to Software engineers. These engineers will thus be able to use the language to define measurement models with ease. The use of general purpose languages to define domain measurement models is thus avoided. Until this moment, no graphic representation permitting a better representation of the model was available.

SSML is a complete language, with a clear syntactic and semantic definition and a solid ontological base. It, moreover, fulfils the following requirements of a DSL: it is usable, it conforms, and it is orthogonal, supportable and simple.

SMML allows users to represent measurement models in various domains.

This language plays a fundamental role in SMF [14] as it allows users to define the measurement models which are the input for the software measurement process. The visual representation of the measurement models mean that SMF is a more usable and intuitive framework for the user. In other words, it makes the measurement process more comfortable.

Among related future works, one important work is that of the extension of SMMM with the Measurement Approach package hierarchy included in the Software Metrics Meta-Model [25].

We shall, moreover, test the usability of the language through a series of experiments based on the ISO 9126 standard. Our study will focus on usability and maintainability. Our idea is to select a group of modeling experts and to test the usability of this new language on them in order to define measurement models.

Finally, we shall apply SMF to real complex environments in order to obtain further refinements and validation.

Acknowledgments

This work has been partially financed by the following projects: INGENIO (Junta de Comunidades de Castilla-La Mancha and Consejería de Educación y Ciencia. PAC08-0154-9262) and ESFINGE (Ministerio de Educación y Ciencia, TIN2006-15175-C05-05).

References

[1] N. Fenton and S. L. Pfleeger, Software Metrics: A Rigorous & Practical Approach, Second Edition: PWS Publishing Company, 1997.

[2] S. G. MacDonell, M. J. Shepperd, and P. J. Sallis, "Metrics for Database Systems: An Empirical Study," in Proceedings of the 4th International Symposium on Software Metrics. IEEE Computer Society, Albuquerque, 1997.

[3] L. C. Briand, S. Morasca, and V. R. Basili, "An Operational Process for Goal-Driven Definition of Measures," IEEE Trans. Softw. Eng, vol. 28, pp. 1106-1125, 2002.

[4] D. Champeaux, Object-oriented Development Process and Metrics: Prentice- Hall, 1997.

[5] M. B. Chrissis, M. Konrad, and S. Shrum, CMMI: guidelines for process integration and product improvement, vol. 1: Pearson, 2006.

[6] ISO/IEC, "Software and Systems Engineering - Guidelines for the application of ISO/IEC 9001:2000 to Computer Software," International Standards Organization, 2004.

[7] J. McGarry, D. Card, C. Jones, B. Layman, E. Clark, J. Dean, and F. Hall, Practical Software Measurement. Objective Information for Decision Makers: Addison-Wesley, 2002.

[8] ISO/IEC, "ISO 15939: Software Engineering - Software Measurement Process.," 2002.

[9] "DSM (Domain-Specific Modeling) Forum Main Page", 2007, http://www.dsmforum.org/.

[10] M. Völter, "A categorization of DSLs", 2006, http://www.voelterblog.blogspot.com/2006/10/categorization-of-dsls.html.

[11] N. A. Allen, C. A. Shaffer, and L. T. Watson, "Building modeling tools that support verification, validation, and testing for the domain expert," in Proceedings of the 37th conference on Winter simulation, pp. 419-426, Orlando, Florida, 2005.

[12] F. García, M. Serrano, J. Cruz-Lemus, F. Ruiz, and M. Piattini, "Managing Software Process Measurement: A Metamodel-Based Approach," Information Sciences, vol. 177, pp. 2570–2586, 2007.

[13] B. Mora, F. Ruiz, F. Garcia, and M. Piattini, "SMML: Software Measurement Modeling Language," Department of Computer Science. University of Castilla - La Mancha Technical Report UCLM-TSI-003, 2008.

[14] B. Mora, F. García, F. Ruiz, M. Piattini, A. Boronat, A. Gómez, J. Á. Carsí, and I. Ramos, "Software Measurement by using QVT Transformation in an MDA context," in Proceedings of the 10th International Conference on Enterprise Information Systems - ICEIS 2008, vol. DISI, pp. 117-124, Barcelona (Spain), 2008.

[15] S. Cook and D. S. Frankel, "Domain-Specific Modeling and Model Driven Architecture," MDA Journal, 2004.

[16] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," ACM Computing Surveys (CSUR), vol. Volume 37, pp. 316-344, 2005.

[17] D. S. Kolovos, R. F. Paige, T. Kelly, and F. A. C. Polack, "Requirements for Domain-Specific Languages," in Proceedings of the First ECOOP Workshop on Domain-Specific Program Development (ECOOP'06), Nantes, France, 2006.

[18] V. Pelechano, M. Albert, M. Javier, and C. Carlos, "Building Tools for Model Driven Development comparing Microsoft DSL Tools and Eclipse Modeling Plug-ins," in Proceedings of the Desarrollo de Software Dirigido por Modelos - DSDM'06 (Junto a JISBD'06), Sitges (Barcelona) España, 2006.

[19] A. v. Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," SIGPLAN Notices, vol. 35, pp. 26-36, 2000.

[20] T. Özgür, "Comparison of Microsoft DSL Tools and Eclipse Modeling Frameworks for Domain-Specific Modeling In the context of the Model Driven Development," in School of Engineering. Ronneby, Sweden: Blekinge Institute of Technology, 2007, pp. 56.

[21] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, "ATL: a QVT-like Transformation Language," 2006.

[22] OMG, "QVT Standard Specification," 2005.

[23] F. Jouault and J. Bézivin, "KM3: a DSL for Metamodel Specification," 2006.

[24] E. Guerra, J. d. Lara, and P. Díaz, "Visual specification of measurements and redesigns for domain specific visual languages," Journal of Visual Languages and Computing, 2008.

[25] OMG, "Architecture-Driven Modernization (ADM): Software Metrics Meta-Model (SMM). OMG Document: dmtf/2007-08-01," Object Management Group 01-08-2007 2007.

[26] F. García, M. F. Bertoa, C. Calero, A. Vallecillo, F. Ruíz, M. Piattini, and M. Genero, "Towards a consistent terminology for software measurement," Information and Software Technology vol. 48 (8), pp. 631-644 2006.

[27] M. Denny, "Ontology building: A survey of editing tools", 2003, http://www.xml.com/pub/a/2002/11/06/ontologies.html.

[28] F. García, M. Piattini, F. Ruiz, G. Canfora, and C. A. Visaggio, "FMESP: Framework for the modeling and evaluation of software processes," Journal of Systems Architecture - Agile Methodologies for Software Production, vol. 52, pp. 627-639, 2006.

[29] M. Feilkas, "How to represent Models, Languages and Transformations?," in Proceedings of the Proceedings of th 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06), pp. 204-213, 2006.

[30] I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez, "Model-based DSL Frameworks," 2006.

[31] G. Canfora, F. García, F. Ruiz, and C. A. Visaggio, "Applying a framework for the improvement of software process maturity," Software: Practice & Experience, vol. 36, pp. 283-304, 2006.

[32] F. García, F. Ruiz, C. Calero, M. F. Bertoa, A. Vallecillo, B. Mora, and M. Piattini, "On the Effective Use of Ontologies in Software Measurement," The Knowledge Engineering Review (in press), vol. 0, pp. 1-24, 2008.

[33] Eclipse, "Eclipse Graphical Modeling Framework (GMF) Main Page", 2007, http://www.eclipse.org/gmf/.

The Transformation-Driven Architecture

Janis Barzdins Sergejs Kozlovics Edgars Rencis
Institute of Mathematics and Computer Science, University of Latvia
Raina blvd. 29, LV-1459, Riga, Latvia
{janis.barzdins, sergejs.kozlovics, edgars.rencis}@lumii.lv

Abstract

This paper proposes a new system building (in particular, tool building) approach, which we call *Transformation-Driven Architecture* (TDA). The basic elements used in this architecture are model transformations, interface metamodels with corresponding engines, and event/command mechanism. The implementation of the UNDO functionality in TDA is also sketched in this paper.

Keywords model transformations, metamodels, events, commands, UNDO

1. Introduction

The increasing popularity of domain-specific languages influenced the appearance of metamodel-based tools such as MetaEdit+ [1], Eclipse GMF [2], AToM^3 [3], Microsoft DSL Tools [4], DiaGen/DiaMeta [5], Pounamu [6], Marama [7], METAclipse [8], and GrTP [9]. Usually, the interaction between the domain and the presentation model is the cornerstone of such tools. The link between these two models may be established by static-mapping approach or by model transformations. However, presentation model is not yet the end product that can be shown to the user. This model should be processed by some engine and displayed in the appropriate form (for instance, as a diagram) to the user. Thus, we get the structure shown in Fig. 1.

According to the Model-View-Controller (MVC) [10] architectural pattern, the domain can be viewed as a model, the presentation — as a view, and the processes that map them — as a controller. If the model and the presentation do not communicate directly, but by means of model transformations, the analogy with the Three-tier-architecture [11] can be noticed. The model corresponds to the data tier, the presentation — to the presentation tier, and the transformations — to the application (or logic) tier. Some other analogies with the Three-tier architecture may also be noticed (see triple-lines in Fig. 1).

In this paper we will go further. The structure from Fig. 1 can be extended by adding additional presentation metamodels and the corresponding engines. The logic that links the domain with all the presentations can be implemented by means of model transformations. In this way a new system building approach arises, which we call *Transformation-Driven Architecture* (TDA).

The following differences between TDA and the approaches used in the abovementioned tools could be realized. Most of the tools have only one presentation that is handled globally by the platform. TDA, in its turn, allows using several different presentations, and each of them can be handled independently by the corresponding engine. Another difference is the wide usage of model transformations in TDA. Transformations can be used not only for linking the domain to one or more presentations, but also for exchanging data between different presentations.

TDA allows transformations to communicate with engines by means of commands and events. Commands are used when engines are called by transformations, while events are used when transformations are called by engines. This differs from the approach used in METAclipse [8], where only events[1] are used: when some presentation event occurs, the platform creates the corresponding object X in the repository and calls the transformation. The transformation may return certain kind of information back to the platform through the same object X. In this approach the transformation is not able to access the platform's functionality directly: it has to use the object X. TDA doesn't have this limitation since both events and commands are used. Not only engines may call transformations, but also transformations are allowed to call engines, which, in their turn, may call other transformations, and so on.

The paper is structured as follows. The next section lists some assumptions setting the background for the explanation of the TDA. Sect. 3 depicts the idea of the TDA and explains the collaboration between transformations and engines. Sect. 4 sketches the solution for implementing the UNDO functionality being a certain issue, which is not implemented (or implemented by storing/loading the model, which is slow) in some tools. In TDA this problem becomes more complicated since not only transformations but also engines have to undo their actions. Although we don't present the whole solution here, we show that the UNDO implementation is possible in TDA. Finally, Sect. 5 concludes this paper.

2. Technical Assumptions for TDA

In this section we list some technical assumptions setting the background for the TDA. The assumptions are as follows.

- The data is stored in some repository (like EMF [12], JGraLab [13] or Sesame [14]) with fixed API (Application Programming Interface).
  Motivation. Fixed API simplifies the way of accessing the repository by engines or by transformations. Fixed API is needed also for UNDO implementation.
- The API of the repository should be available for one or more high-level programming languages (such as C++ or Java), in which presentation engines will be written.
  Motivation. This allows the engines to be able to exchange the data with transformations through the repository.
- Model transformations may be written in any language (for instance, any textual language from the Lx family [15] or the graphical language MOLA may be used [16]). However, the transformation compiler/interpreter should use the same repository API as the engines.
  Motivation. This will simplify the UNDO implementation.
- When a transformation is called, its behavior depends only on the data stored in the repository.
  Motivation. This will also simplify the UNDO implementation.

[1] They are called "commands" in METAclipse, while in our context they play the role of events.

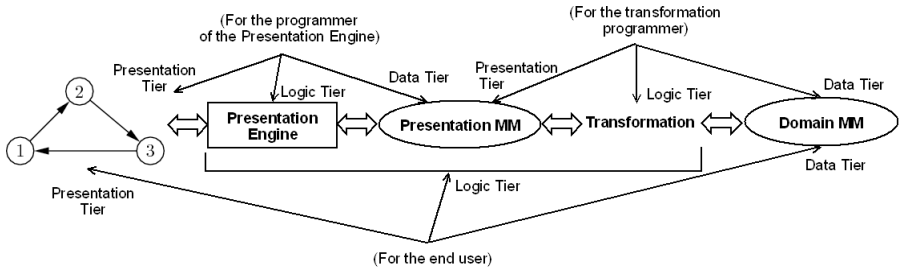


Figure 1. The chain between the end user presentation and the domain. Triple-lines show relations to the three-tier architecture.

If the transformations needs some internal variables, these variables may be stored in the repository as well.

- The metamodels do not change at runtime. All the changes in the repository are at the instance-level.
  Motivation. This is a simplification only. The changes at the metamodel-level can be considered in the same way as changes at the instance-level, if needed.
- Only one repository is used. Interaction between different repositories as well as distributed data storage are not considered.
  Motivation. This is also a simplification. A transparent super-API over different/distributed repositories may be defined, if needed.
- Only one module (transformation or engine) is allowed to access the repository at the same time. Concurrency and locking issues are not considered.
  Motivation. On the one hand, concurrency and locking issues are complex enough and require separate research. On the other hand, most metamodel-based tools don't have concurrent access to the repository, and, usually, it is not so essential.

3. The Transformation-Driven Architecture

Fig. 2 shows a sample system that depicts the essence of the TDA. The cloud in the center is the system metamodel that may be divided into several parts. In our example, one can notice parts called "Domain metamodel" and "Presentation metamodel". The presentation metamodel has the corresponding engine, and, in fact, the whole chain from Fig. 1 can be found also in Fig. 2. There are also other presentation metamodels (the dialog metamodel, the database metamodel and the XML metamodel in our example) with their engines. Each presentation metamodel can be considered as an *interface metamodel* for the corresponding engine.

Let's call the remaining part of the system metamodel (to which the class "Command" belongs) the core part. The five parts mentioned above are linked to this core part, thus complementing it. The core part also has its own engine called "Head engine". The head engine may be considered as an "operating system" (OS), and the other engines as device drivers. The real functionality is performed by model transformations that may be considered as programs.

3.1 How Does the System Work

In the beginning the control is gained by the head engine. It constructs the main window, where the user can open an existing project or create a new one. In the first case the data is simply loaded to the repository. In the second case the head engine calls the corresponding transformation that fills the repository with some initial data.

When the project has been initialized, each engine may generate events, and the corresponding transformation being able to handle this event is executed. Before calling the transformation, the engine in which the event occurs creates an instance of the corresponding "Event" subclass. The properties (attributes and links) of this instance may be considered as arguments for the transformation.

While events are used to call transformations from engines, commands are used for the opposite direction — to call engines from transformations. When there is a need to call an engine, the transformation creates a command and asks the head engine to execute this command like a program may call an OS function to get access to some device. The head engine determines which of the engines must be called and passes the control to it. That is like the OS passes the control to the corresponding device driver.

Each command is an instance of some "Command" subclass. For example, a command for the Presentation Engine may be of the type "PresentationCommand". The presentation engine may have several command types that are descendants of the class "Command" and that belong to the presentation metamodel, so the appropriate type must be selected and the instance created (see also [9]).

In order to execute the given command, the head engine checks the type of the corresponding instance and determines the part of the metamodel to which it belongs. After that the corresponding engine is called.

While a command is being executed, events may be created and transformations may be called. These transformations may also create commands. While existing commands are being executed, new commands have to be managed correctly. Thus, the appropriate data structure is required to store the commands. We call it the command queue, however, in reality it is a hybrid of the queue and the stack.

3.2 The Command Queue

Assume that during execution of some command other commands are not created. In this case the command queue is a real queue. Let there be a fictive command EOC ("End of Commands") denoting the end of the command queue (EOC is the command after the last real command). We assume there is a pointer to this EOC instance. When a transformation needs to add a command, it finds the EOC and inserts the new command just before the EOC by creating the corresponding "previous"/"next" links. When the head engine processes the queue, it finds the first command and executes the commands starting from the first until the EOC is reached.

However, if execution of a command creates other commands, the new commands should not be added to the end of the queue. Fig. 3(a) depicts the point. Assume that during the execution of command A the transformation, which adds additional commands A_1 and A_2, is called. A is still being executed. So, A_1 and A_2 should be considered as parts of A. Thus, A_1 and A_2 have to be executed before B, not after C as it would be if A_1 and A_2 were added just before the EOC.

In order to solve this problem, we modify the command queue slightly. Before starting to execute A, it is replaced by EOC, and

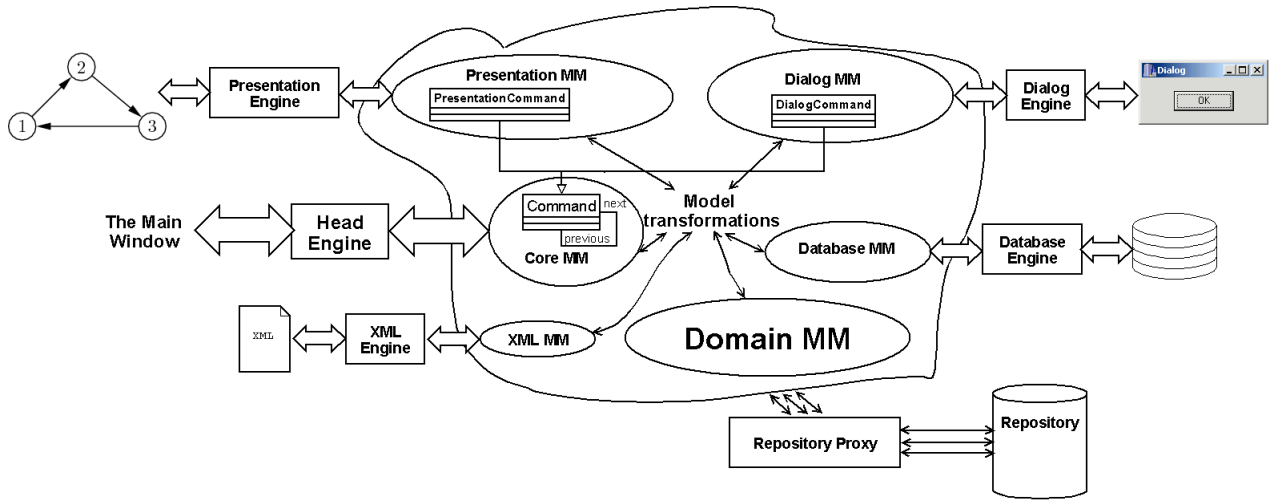


Figure 2. An example revealing the essence of the Transformation-Driven Architecture.

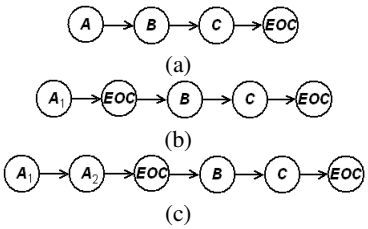


Figure 3. (a) A sample command queue. (b)–(c) While processing command A two more commands are added — A_1 and A_2.

the EOC pointer is moved to this new EOC. Now, we can execute A. If new commands need to be added, they are added before the just created EOC, and, thus, also before B (see Fig. 3(b) and (c)).

So, the command queue is actually a stack of queues, where the queues are separated by EOCs (see Fig. 4).

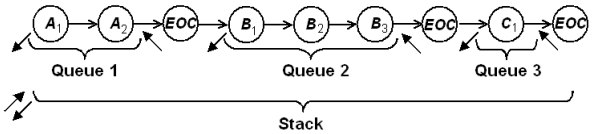


Figure 4. The structure of the command queue (the "stack of queues").

4. The Implementation of UNDO

An important question arises whether it is possible to implement the UNDO functionality in TDA. It is worth to note that during the UNDO not only a) the actions performed by transformations have to be undone, but also b) the engines must be returned to the corresponding states.

The answer to the question just mentioned is "Yes". The problem a) can be solved by accessing the repository through the proxy that manages the UNDO history. The problem b) is solved by introducing additional functionality that can be used to notify the proxy when the engines change their states. We sketch these solutions below.

In order to solve the problem a), some repository API functions are hooked by the repository proxy. That is why the API has to be fixed.

The repository API contains read-only functions (such as functions for traversing the repository) as well as functions that modify the repository (modificating functions). Modificating functions can be divided into primitive and non-primitive ones. We assume the following instance-level functions to be primitive:

- adding/deleting an instance;
- adding/deleting a link between two instances;
- changing the value of some attribute (property).

(We could select also certain metamodel-level routines if the metamodel could be changed at runtime.) It is considered that non-primitive modificating functions can be implemented by means of primitive functions. For instance, the function for cascade delete (that deletes the aggregate with its parts recursively) can be implemented through calls to the function for deleting single instance and to the function for deleting single link. This allows to handle all repository changes by considering only primitive functions. Since engines and transformations use the same API, hooking primitive functions handles all the changes made in the repository.

When a primitive function is called, the information needed to undo and also to redo the corresponding operation is being written into the history. The history consists of *transactions*. Each transaction contains undo/redo information for several actions. Transactions are started before certain events like "New box" or "New line" events in the presentation engine.

The solution for the problem b) involves notifying the proxy about the changes in the states of one or more engines. Each engine may have several units for which UNDO can be called independently. We call these units *diagrams*. For instance, there may be several open graph diagrams managed by the Presentation Engine. If the user had changed Diagram A before Diagram B, then UNDO in Diagram A should not affect Diagram B (unless the diagrams depend on each other). For each diagram the engine has to know how to track the changes and to undo them. For instance, in graph diagrams the coordinates of objects may be saved, and, when UNDO is called, restored. The issues concerning what the diagram is and how its states are saved and restored depend on the engine.

For UNDO we are interested not in engines, but in diagrams that have been changed (since during the same transaction some diagrams of the same engine may be changed, and some may be left unchanged). The function `DiagramChanged(DiagramID, CurrentStateID)` is used to notify the proxy about diagram changes. The diagram is being changed from the previous state to the state `CurrentStateID`. `DiagramChanged` should also be called for each diagram to specify initial states in the beginning. It is assumed that the diagram change corresponds to the current transaction. Thus, undoing this transaction will return the diagram to the state before the change.

The list of other UNDO-related functions is presented below (but without implementation details):

- `CreateCheckPoint()` — creates a new transaction; should be called before certain events.
- `CanUndo(DiagramID)` — returns `true` iff the last transaction that modified the given diagram can be undone.
- `Undo(DiagramID)` — undoes the last transaction where the given diagram has been modified (let's call this transaction T_1). Other transactions may also be undone. For instance, if another diagram has been modified in T_1 and also in transaction T_2, where T_2 comes after T_1, then T_2 also needs to be undone. Thus, other diagrams may need to change their states as well. For that reason, `Undo` returns the list of diagrams and the corresponding states.
- `CanRedo(DiagramID)` — returns `true` iff there was an undone transaction where the given diagram has been changed. This transaction can be redone.
- `Redo(DiagramID)` — acts similar to `Undo`, but in the opposite direction.
- `ClearHistory(DiagramID)` — deletes from the history all the transactions that affected the given diagram. Some other transactions may also be deleted to keep the history consistent.
- `ClearAllHistory()` — clears all UNDO history.

In fact, functions for creating dependencies between transactions may also be introduced. For instance, the function `ObjectMustExist(ObjectID)` may be used to indicate that undoing the transaction where the given object has been created forces undoing also the current transaction.

5. Conclusion

The main contribution of the paper is the idea of using several presentation metamodels with the corresponding engines, where the connection between all the metamodels (including the domain metamodel) is ensured by model transformations. The command queue has been introduced as the way of transferring control between transformations and engines. The basic ideas of the UNDO implementation have also been presented with the aim to show that such an implementation is possible. A detailed explanation and comparison to other approaches for UNDO are subject to further research.

In this paper, we haven't addressed the problem of writing model transformations to be used with TDA. One of the solutions is to write transformations from scratch each time a new system is being built. Another solution is to write a universal transformation, which would handle many typical tasks arising in the tool building process. A special tool definition metamodel may be introduced, and the universal transformation may interpret instances of this metamodel. This research topic is also of our interest.

Some ideas presented in this paper have been successfully implemented in the recent version of transformation-based tool building platform *GrTP* [9]. There are two presentation engines (the graph diagram engine and the dialog engine), which communicate with model transformations by means of commands and events. Commands are stored in the command queue, and there are cases when the command queue has to be modified by introducing EOC command (for instance, when one modal dialog calls another). The transformation-driven architecture has proved to be powerful enough to build an experimental tool — an editor for UML class diagrams with profiles (stereotypes) and other advanced features. Moreover, a graphical query tool for RDF databases has also been built using this architecture [17].

It is a well-known fact that the development of presentation engines is a very time-consuming process. That's why it is done only once in our approach. On the contrary, it is relatively easy to write transformations working with metamodels. And this is the basic profit we can achieve by means of TDA.

Acknowledgments

The authors would like to thank Karlis Cerans, Renars Liepins and Sergejs Rikacovs for their help and suggestions.

References

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

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

[3] J. de Lara, H. Vangheluwe and M. Alfonseca. Meta-Modeling and Graph Grammars for Multi-Paradigm Modeling in AToM3. *Software and System Modeling*, 3(3), 2004., pp. 194-209.

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

[5] DiaGen/DiaMeta. `http://www.unibw.de/inf2/DiaGen/`

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

[7] J. Grundy, J. Hosking, N. Zhu1 and 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.

[8] A. Kalnins, O. Vilitis, E. Celms, E. Kalnina, A. Sostaks and J. Barzdins. Building Tools by Model Transformations in Eclipse. *Proceedings of DSM'07 Workshop of OOPSLA 2007*, Montreal, Canada, Jyvaskyla University Printing House, pp. 194-207.

[9] J. Barzdins et al. GrTP: Transformation Based Graphical Tool Building Platform. *Proceedings of MDDAUI'07 Workshop of MODELS 2007*, Nashville, Tennessee, USA.

[10] T. Reenskaug. The Model-View-Controller (MVC). Its Past and Present. *JavaZONE*, Oslo, 2003. JAOO, Arhus.

[11] Wayne W. Eckerson. Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications. *Open Information Systems 10*, 1 (January 1995): 3(20).

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

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

[14] Sesame. `http://www.openrdf.org`.

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

[16] A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA, *Proceedings of MDAFA 2004* (Model-Driven Architecture: Foundations and Applications 2004), Linkoeping, Sweden, pp. 14–28.

[17] G. Barzdins, E. Liepins, M. Veilande, M. Zviedris. Semantic Latvia Approach in the Medical Domain. *Proceedings of the 8th International Baltic Conference* (Baltic DB&IS 2008), Tallin, Estonia, pp. 89–102.

When Frameworks Let You Down

Platform-Imposed Constraints on the Design and Evolution of Domain-Specific Languages

Danny M. Groenewegen Zef Hemel Lennart C. L. Kats Eelco Visser
Delft University of Technology, The Netherlands
{d.m.groenewegen,z.hemel,l.c.l.kats,e.visser}@tudelft.nl

1. Introduction

Application frameworks encapsulate knowledge of a particular domain in a reusable library. However, based on a general-purpose language, these do not provide notational constructs for the particular domain, and are limited to the static checks of the host language. Verification of correctness, security, and style constraints, and optimizations in terms of the application domain are not possible or very hard. It is common practice to build a Domain-Specific Language (DSL) as a thin abstraction layer over a framework, providing domain-specific notations and enabling analysis and reasoning at the level of these constructs (5; 6).

Using an established application framework as the platform for a DSL helps in the understanding the domain, and supports reuse of the domain knowledge gathered by the framework developers. By means of a *reference application*, a small application implemented using the framework, it is possible to identify the basic operations that are important for the domain, and map these to language constructs. Rather than designing a complete DSL "on paper," before its implementation, it is good practice to incrementally build higher-level abstractions on top of the basic operations through a process of inductive design (7). This enables quick turn-around time for the development of the DSL and the subsequent gradual extension as new applications are found, and new insights into the domain are acquired.

Application frameworks abstract over a lower-level platform. For instance, the Java Persistence API (JPA) abstracts over persistence operations in relational databases. As frameworks are designed for direct use by programmers, they tend to avoid complex interfaces, hiding underlying details, and limiting the number of concepts a programmer has to deal with. For code generation, these hidden details may become important as a DSL evolves and new features are added to it. A domain-specific language can "outgrow" its platform if it does not provide the elementary operations required to implement new aspects of the language. Despite the benefits of an inductive approach towards designing DSLs, and the merits of using established frameworks, this seems to be a recurring problem in the evolution of DSLs. In this paper, we give examples of occurrences of this problem and possible scenarios that describe how it can be dealt with. The examples we give are taken from our own experience with WebDSL, a domain-specific language for dynamic web applications with a rich data model (7). However, we believe these issues occur in other DSLs as well and therefore pose a more common problem.

2. Examples

The original design of WebDSL largely followed that of its target platform, the Seam web development framework. However, over time, new features were introduced that did not always match with this framework. In this section we give three examples of such extensions that were not directly or sufficiently supported by the framework.

Data Model Modularity WebDSL has first-class language constructs for data model definitions in the form of entity declarations. Data model definitions can be modularized into modules that each represent a different application concern. For example, consider a conference management system that has a generic user management system, defining usernames and passwords. In this system, all users have a list of papers they authored associated with them. Rather than directly modifying the data model for users, entangling the definition with a new concern, we designed a mechanism to augment already defined entities in other modules (see Figure 1). However, these kinds of partial entity declarations do not naturally translate to regular Java classes. They would instead require partial classes (such as in C#) or inter-type declarations (such as in AspectJ), but neither is supported by the current target platform.

Template Mechanism For reuse of page or page element definitions, WebDSL provides a template mechanism. Every page may import one or more templates, and these in turn may import other templates. For increased flexibility, we designed a system based on dynamically scoped template overrides. This template mechanism is inspired by the TEX typesetting system (4), which provides dynamically scoped macro definitions. Consider Figure 2, which demonstrates customizing the default behavior of templates with local template redefinitions. Unfortunately, Seam's templating mechanism (Facelets) is based on the idea of template inheritance, and is incompatible with the dynamic scoping mechanism. Again, this extension posed a mismatch between what is supported by the framework and what we would like to support in the DSL.

Access Control WebDSL provides specialized constructs for access control. While basic access control could be implemented

```
module usermanagement

  entity User {
    username :: String
    password :: Secret
  }

module paper

  extend entity User {
    authored → Set<Paper>
  }
```

Figure 1. Extension of data models in different modules

```
module pagelayout

  define main() {
     header()
     body()
     footer()
  }

  define body() {
     // Default body
  }

  define page home() {
    main()

    define body() {
      // New body, defined in the scope of page home()
    }
  }
```

Figure 2. Dynamically scoped templates.

```
module ac

  rule page viewUser(*) {
    viewingAllowed()
  }

module userpages

  define page viewUser(u : User) {
    // if viewingAllowed()
    output(u.name)
    output(u.authored)
  }

  define page userList() {
    for (u : User) {
      // if viewingAllowed()
      navigate(viewUser(u))
    }
  }
```

Figure 3. Modular access control rules.

through the use of if-statements, which hide certain (parts of) pages from the user if they do not have access to it. This approach, however, would be rather low-level, and would cause tangling of the concern with other aspects of the code.

The Seam framework offers an access control solution, which was deemed too inflexible for use in WebDSL. For instance, it assumes role-based access control policies, while WebDSL should support other policies as well. Seam's access control policy does not take access restrictions into account for the presentation of web pages, and it is difficult to extend with additional behavior. In WebDSL we wanted to support access control as a separate aspect that is woven into the application such that access control rules can have direct impact on page contents, e.g. by hiding navigation links to restricted pagers. Therefore, the language was extended with a specialized access control sub-language that allows the specification of access control as a separate set of rules (1). These rules put access control restrictions on pages, templates or other high-level components (see Figure 3), which serve as join points for the access control aspect language.

3. Scenarios

The examples described in the previous section indicate clear mismatches between frameworks and DSLs. In this section we describe a number of scenarios that can be explored to deal with these issues.

3.1 Introducing Intermediate Transformation Steps

Some platform mismatches can be dealt with by introducing model-to-model transformations, resolving certain shortcomings of the platform. For instance, dynamically scoped templates can be implemented through a transformation that statically expands templates. Template calls can be resolved statically and replaced by the template contents at generation time. A drawback of this solution is that it leads to a significant increase in the size of the generated code and cannot support recursive template calls. DSL features that require a type of aspect weaving can also be implemented through model transformations. Access control checks are woven into the pages they apply to, so that all that is left is DSL code that can be directly mapped to the underlying framework (1). Strictly adhering to the implementation style prescribed by the framework, these model-to-model transformations can cause abstraction inversion in the generated code. A feature such as templating is reimplemented on top of the existing mechanism, possibly incurring a performance penalty.

Likewise, if a target language does not suffice for a particular task, because it lacks certain features, it is possible to introduce these as extensions to the language. In particular, features for modularity such as partial classes and methods or multi-methods can be beneficial for code generation, and are not supported for all languages. By treating the target language as a first-class model, rather than plain text, it is possible to use the same techniques for model transformations as used in the rest of the DSL for realizing these extensions (2). In this manner the WebDSL generator internally generates partial Java classes, which in a later step of the generator are merged through a set of model transformations.

3.2 Adapting the platform

Another option is to adapt the application framework to better support the DSL. The JavaServer Faces library could be adapted to support WebDSL's dynamic scope templates for instance, and Seam's access control framework could be extended to suit our particular needs.

However, the developers of the DSL should be very well acquainted with the platform they are adapting. Often the developers are not motivated to adapt other people's work (the "not invented here" attitude).

Once changes have been made to the target platform, the original developers of that platform may evolve it over time (assuming the framework developers are not the same group as the DSL developers). In this case there are three options: either stick to the original adapted target platform, apply the changes made to the old version to the new one, or try to convince the framework developers to adopt the changes. Clearly, this is a maintenance issue.

3.3 Switching to a Different Target Platform

If a given platform really does not suffice for a particular task, it may also be an option to replace it with an alternative. This can be another, similar framework, or may be a lower-level alternative, for instance by generating plain Java Servlets code.

The approach of compilation by normalization (3) can be used to guide the generation to a low-level target. It iteratively abstracts from a low-level target platform, while not restricting access to the underlying primitives.

One issue that may come up is having introduced leaky abstractions. The DSL was initially built as a thin layer on top of a framework, so is likely to be fairly specific to that platform. It may be very difficult or even impossible to replace the platform with another one, because it would imply changes to the DSL.

When it is decided to generate lower-level code it is likely that much more code has to be generated, mimicking the behavior of the framework that was previously used, because the semantics of the DSL should not change. Thus, developing the generator is likely to require a larger effort than when targeting a high-level framework. Frameworks often encapsulate years of experience in their particular domain. Replacing them with a home brewed framework should not be underestimated, neither in terms of maintenance or performance optimizations. On the other hand, frameworks are usually optimized to make writing code using them as simple as possible for the developer, often at the expense of performance. For instance, many Java frameworks depend on reflection to reduce the size of client code. However, this incurs a runtime overhead, which can be avoided when code is generated.

3.4 Cutting Your Losses

It may turn out that the previously described scenarios are simply too expensive. The time and effort required to switch to a different platform, adapt it, or to add additional model transformations can be too costly and the gain too small. Settling for Facelet templates with inheritance, and the somewhat inflexible access control solution of Seam could be an acceptable solution if this is the case. In this scenario, the DSL and the framework are kept in sync. No features are added to the DSL unless fully supported by the framework.

This raises the question of what expectations one has of DSLs. Are they intended as only a thin layer of syntax and set of checks on top of a framework, or is that just a point of departure that leads to an independently evolving language? Our experience from designing and implementing WebDSL has taught us that frameworks form a very useful foundation to build upon, but that at some point the DSL may simply outgrow its underlying platform. In this paper we outlined four scenarios to deal with this problem, each having their respective advantages and drawbacks. Future research is required to reveal best practices in this area.

Acknowledgments This research was supported by NWO/JACQUARD projects 638.001.610, *MoDSE: Model-Driven Software Evolution*, 612.063.512, and *TFA: Transformations for Abstractions*.

References

[1] D. Groenewegen and E. Visser. Declarative access control for WebDSL: Combining language integration and separation of concerns. In D. Schwabe and F. Curbera, editors, *International Conference on Web Engineering (ICWE 2008)*. IEEE CS Press, July 2008.

[2] Z. Hemel, L. C. L. Kats, and E. Visser. Code generation by model transformation. A case study in transformation modularity. In J. Gray, A. Pierantonio, and A. Vallecillo, editors, *International Conference on Model Transformation (ICMT 2008)*, Lecture Notes in Computer Science. Springer, June 2008.

[3] L. C. L. Kats, M. Bravenboer, and E. Visser. Mixing source and bytecode. A case for compilation by normalization. In G. Kiczales, editor, *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2008)*, Nashville, Tenessee, USA, October 2008. ACM Press.

[4] D. E. Knuth. *The TEXbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[5] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[6] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[7] E. Visser. WebDSL: A case study in domain-specific language engineering. In R. Lammel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Lecture Notes in Computer Science. Springer, 2008. Tutorial for International Summer School GTTSE 2007; to appear.

Visual Specification of a DSL Processor Debugger

Tamás Mészáros
Budapest University of Technology and Economics
Department of Automation and Applied Informatics
mesztam@aut.bme.hu

Tihamér Levendovszky
Budapest University of Technology and Economics
Department of Automation and Applied Informatics
tihamer@aut.bme.hu

Abstract

Graph rewriting-based model transformation is an essential tool to process graph-based visual models. If the execution of transformations is not supported by the continuous presentation of the modifications performed on the model, the traceability and the debugging of transformations becomes difficult. Recent modeling tools usually support the definition of rewriting rules based transformations in a visual or textual way, and only a few of them support visual debugging facilities. These debuggers are hand-coded at a price of a huge amount of work. This paper presents a model transformation debugger built on the top of the animation framework and the transformation engine of the Visual Modeling and Transformation System. The integration of the transformation engine and the animation of the user interface are described with visual modeling techniques.

Categories and Subject Descriptors I.6.2., I.6.3. [**Simulation and Modeling**]: Simulation Languages, Applications

General Terms Design, Languages

Keywords *Model transformation, Animation, VMTS*

1. Introduction

Domain-specific modeling is a powerful technique to describe complex systems in a precise but still understandable way. The strength of domain-specific modeling lies in the application of domain-specific languages to describe a system. Domain-specific languages are specialized to a concrete application domain; therefore, they are particularly efficient in their problem area compared to general purpose languages.

Models created with such languages usually need further automated processing methods to utilize the information expressed by the models in real, end-user applications. The processing may be similar to the source code compilers which convert human-readable source code to byte-code or machine code executed by the hardware or a virtual machine, but various model-to-model transformations are also frequent.

When developing a model processor for a language, it is important to be able to efficiently trace and debug the operations performed by the processor. It is not negligible how much effort is required to develop a visual debugger either. The motivation of our work is to provide a model transformation debugger solution built with the help of visual modeling techniques.

Visual Modeling and Transformation System (VMTS) [1] is a general purpose metamodeling environment supporting an arbitrary number of metamodel levels. Models in VMTS are represented as directed, attributed graphs the edges of which are also attributed. The visualization of models is supported by the VMTS Presentation Framework (VPF) [2]. VPF is a highly customizable presentation layer built on domain-specific plugins which can be defined in a declarative manner.

VMTS Animation Framework

The VMTS Animation Framework (VAF) [3] is a flexible framework supporting the real-time animation of models both in their visualized and modeled properties. The architecture of VAF is illustrated in Figure 1.

VAF separates the animation of the visualization from the dynamic behavior (simulation) of the model. For instance, the dynamic behavior of a graphically simulated statechart is really different from that of a simulated continuous control system model. In our approach, the domain knowledge can be considered a black-box whose integration is supported with visual modeling techniques. Using this approach, we can integrate various simulation frameworks or self-written components with event-driven communication. The animation framework provides three visual languages to describe the dynamic behavior of a metamodeled model, and their processing via an event-driven concept. The key elements in our approach are the *events*. Events are parametrizable messages that connect the components in our environment. The services of the Presentation Framework, the domain-specific extensions, possible external simulation engines (*ENVIRONMENT* block in Figure 1) are wrapped with *event handlers*, which provide an event-based interface. Communication with event handlers can be established using events. The definition of event handlers is supported with a visual language. The visual language defines the event handler, its parameters, the possible events, and the parameters of them - called *entities* (*Event handler model* in the figure). The default implementation of an event handler is generated based on the model, but the event handler methods which interact with the wrapped object have to be written manually (Implementation block).

The animation logic can be described using an event-driven state machine, called *Animator* (*Animator state machine* block*)*. We have designed another visual language to define these state machines. The state machine consumes and produces events. The transitions of the state machine are guarded by conditions (*Guard* property) testing the input events and fire other events after performing the transition (*Action* property). States also define an *Action* property, which describes an operation that is executed when the state becomes active. The input (output) events of the state machine are created in (sent to) another state machine or an event handler. The events produced by the event handlers and the state machines are scheduled and processed by a DEVS [4] based simulator engine (*Animation Engine*).

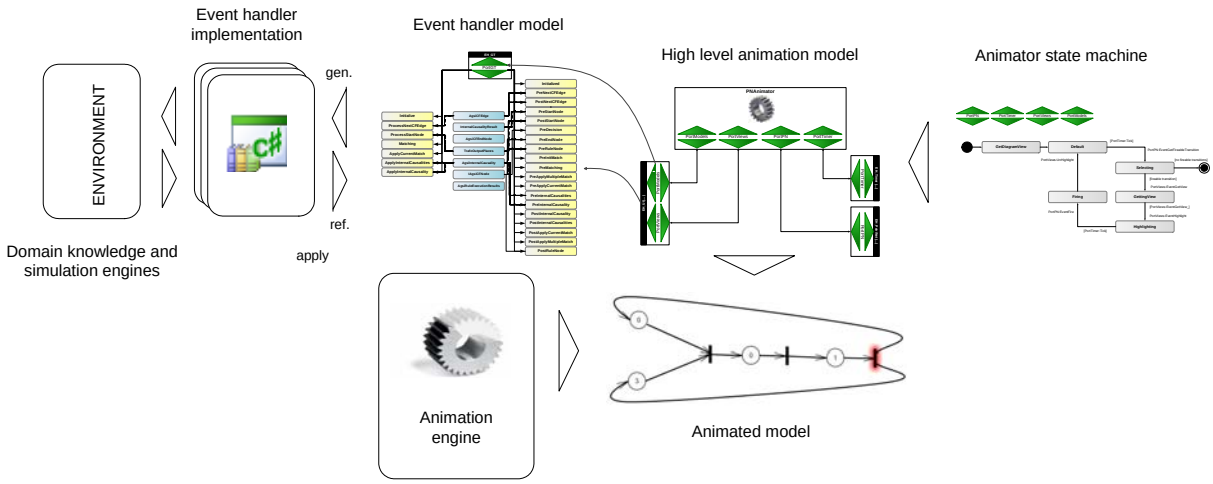


Figure 1. Architecture of the VMTS Animation Framework

The event handlers and the state machines can be connected in a high-level model (*High level animation model*). The communication between components is established through ports. Ports can be considered labeled buffers, which have a configurable but predefined size.

On executing an animation, both the high-level model and the low-level state machines are converted into source code, which is executed after an automated compilation phase.

Graph rewriting

Recall that in VMTS, models are represented as directed, attributed graphs. Model elements are represented by nodes and the connections between the elements are defined by the edges of the graph. This representation facilitates the applications of various graph transformation algorithms. Graph rewriting [5] is a powerful technique for applying graph transformations. Graph transformation consists of rewriting rules. Each rewriting rule has two parts: a Left Hand Side (LHS) and a Right Hand Side (RHS). The LHS defines a model pattern which has to be found in the input model, while the RHS describes a substitute pattern the match of the LHS has to be replaced with. Editing graph rewriting rules is supported via the Rule Editor plugin of VMTS. The execution order of rewriting rules can be defined with the help of the Visual Control Flow Language [6]. A Visual Control Flow model may contain six types of elements: *Start* node, *End* node, *Rule* node, *Decision* node, Flow edge and *External causality* edge. The *Flow* edge indicates the direction of the control flow. The *Start* node defines the entry point of the transformation, it also specifies the output model (if different from the input model). The *End* node indicates the end of the transformation. The *Rule* node means the application of a rewriting rule, which is defined in another model, and the *Rule* node only references that model. The *Decision* node is used to branch in the flow based on a predefined Object Constraint Language (OCL) [7] condition. The external causality edge can declare that an element on the LHS of a rule matches another element on the RHS of another rule. The operation described by a rewriting rule is called *internal causality* in our terminology. There are three types of internal causalities: (i) *create*, which is used to create new elements in the output model; (ii) *modify*, which is appropriate for changing the attributes of the matched elements and (iii) *delete*, which deletes a specified subset of nodes matched on the LHS. The *create* and *modify* causalities are defined using the Imperative OCL [8] language.

The application of a rewriting rule usually consists of two main steps: (i) searching a subgraph (match) in the input model that matches the LHS pattern of the rule, (ii) execution of the rewriting rules. If the *Exhaustive* attribute of the rewriting rule is set to true, then the same rule is applied until no match can be found. Otherwise, the next rule along the control flow is applied.

2. A DSL Processor Debugger

The aim of building a debugger for visualizing model transformations is to be able to trace the transformation process, and to have the possibility to intervene at runtime. Thus, we had the following objectives before beginning to design the debugger: (i) the input and output models should be visualized and should always reflect the current state of the models; (ii) the control flow model should be animated to be able to exactly trace the execution of the transformation; (iii) the actually executed rewriting rule should also be shown and in case of a successful match, the match should be visualized; (iv) the transformation should run step-by-step and continuously, the continuous running should be able to be interrupted by breakpoints, and the user should be allowed to perform jumps in the control flow, (v) it would also be welcome, if the models (at least the host and the target) could be edited at runtime.

Event handler model

The model of an event handler defines the events it can handle, the parameters of the events, and the interface of the event handler. The interface of a component is described by a set of *ports*: both event handlers and state machines provide their services through ports which can be connected to each other.

Before implementing the animation logic with event-driven state machines, we had to wrap our graph transformation engine (the "ENVIRONMENT" in this case) with an event handler, to provide an event-driven uniform interface for the animators. However, after performing the wrapping, we can use this event handler not only for the debugging solution, but also for various other simulations requiring graph rewriting-based model transformation.

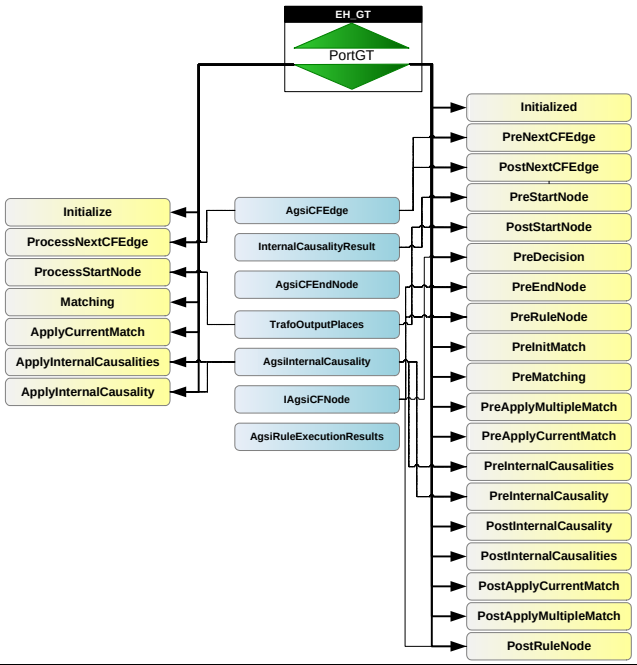


Figure 2. Model transformation engine event handler

Figure 2 illustrates the event handler model of the model transformation engine. The event handler (*EH_GT*) defines one port (*PortGT*) to send and receive events. On the left hand side the events received by the event handler can be seen, on the right hand side the events sent by the event handler are presented. In the middle, the entities (the parameters used by the events) are enumerated. The events sent by the event handler usually begin with "*Pre*" or "*Post*". The pre-events are fired before performing a specific operation, whereas the post-events are fired afterwards. After sending a pre-event, the event handler usually waits for another event to instruct the transformation engine to perform a step. Thus, we have the possibility to skip an operation or to modify its parameters. We have defined a pre-post event pair for each type of element in the transformation control flow: *Pre/PostNextCFEdge*, *StarNode*, *EndNode*, *Decision* and *RuleNode*. These events are also parametrized with the classes of the model transformation engine. The *Pre/PostNextCFEdge* events have a parameter of type *AgsiCFEdge* which points to the flow edge in the control flow. After sending a *PreNextCFEdge* event, a *ProcessNextCFEdge* event has to be sent to the event handler to follow the edge. The *ProcessNextCFEdge* event has a parameter of type *AgsiCFEdge* as well. This parameter should point to the edge to follow, thus, by pointing to an edge other than the one used by the *PreNextCFEdge* event, we can jump to an arbitrary edge in the control flow.

Rule nodes in the control flow are processed in the following steps: (i) The matcher algorithm searches for matches according to the LHS of the rule. If the rule node is configured for multiple matches, then several matches are found. Parts of the matches can also come from external causalities. (ii) The internal causalities of the rewriting rule are executed on the first match resulting in that several model elements may be deleted or created. (iii) In case of a multiple match, the following match is selected, and (ii) is performed again. (iv) In case of an exhaustive match, the complete process is repeated from (i) until no match can be found. The individual steps of this process are also wrapped with events, we have created the pre/post versions of *RuleNode*, *ApplyMultipleMatch*, *ApplyCurrentMatch*, *ApplyInternalCausalities*, *ApplyInternalCausality* events. The *PreMatching* event is sent before starting the matching phase, the *PreInitMatch* event is sent before initializing the match with the elements coming from external causalities. Influence on the matching and rewriting phase is also provided: the *PreApplyCurrentMatch* is fired before applying a match, however, one can override this match by sending an *ApplyCurrentMatch* with parameters different from the ones in the *PreApplyCurrentMatch*. We can also override the set of applied internal causalities and each internal causality as well with the help of the *ApplyInternalCausalities* and the *ApplyInternalCausality* events. The event flow of the rewriting phase is illustrated in Figure 3. The sequence diagram depicts the events fired between the transformation event handler and the animation engine when applying a rewriting rule including several causalities of it. This sequence diagram is included here for illustration purposes, not actually modeled, it is distilled from the state machine models, and only its implementation is generated.

Animation model

The animation can be described with the help of another visual language which can model state machines. These state machines communicate via events: the state transitions trigger the existence of a specific event on a specific port (or a specific event combination on a set of ports), and fire events when performing the state transition. The state machine is called *Animator* in our terminology. Animators are modeled on two levels: (i) on the high-level representation several animators and event handlers can be connected, and their interaction can be modeled, (ii) on the low level representation the individual states and transitions between the states of the state machine can be modeled.

Figure 4 illustrates the composition of animators and event

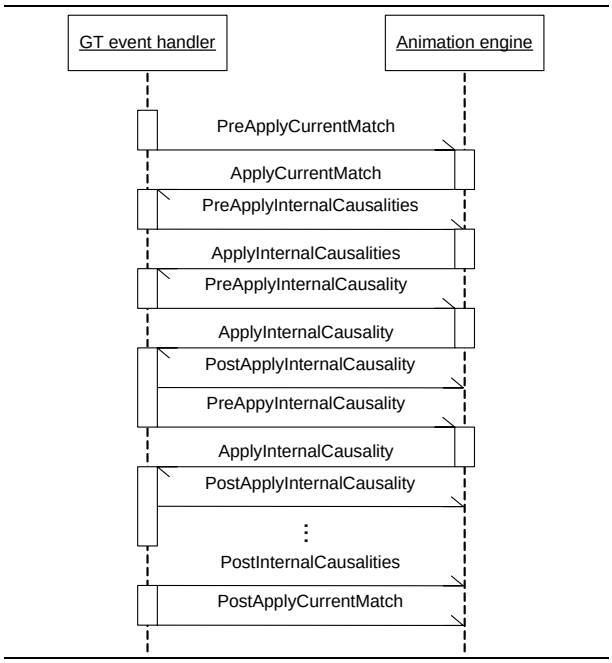


Figure 3. Event-flow of the rewriting phase

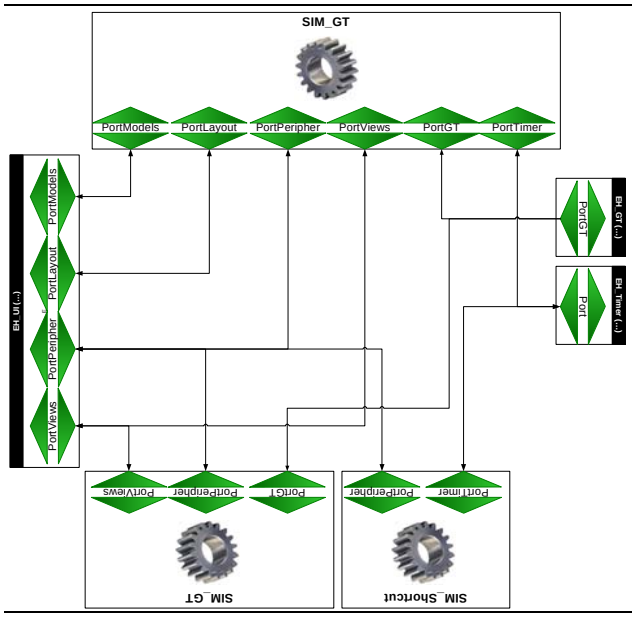


Figure 4. High level animation model of the debugger

handlers which implement the model transformation debugger. Event handlers (*EH_UI*, *EH_GT*, *EH_Timer*) can be seen on the left and right sides of the figure. The high level representation of three animators (*SIM_GT*, *SIM_MatchHighlighter*, *SIM_Shortcut*) is depicted on the top and the bottom of the figure.

The *EH_UI* element references the UI event handler which wraps the user interface API of VMTS and the model management methods. The *EH_Timer* element points to the event handler of a real-time clock, which fires *Tick* events with a predefined frequency. The frequency of the timer is set through the *Frequency* parameter of the event handler to 500 msec, thus one step is performed every half second in continuous execution of the transformation.

In Figure 4, one can see three animators: *SIM_GT*, *SIM_MatchHighlighter* and *SIM_Shortcut*. *SIM_GT* animates the control flow model, initiates the execution of the match and rewriting operations. *SIM_Shortcut* catches the key-presses, and instructs the *EH_Timer* to fire a *Tick* event if the F11 key was pressed. This feature is useful, if the timer is paused, and the user can execute the transformation step-by-step by hitting the F11 key. *SIM_MatchHighlighter* catches the mouse events, and highlights matched and created elements in the host and the output model of the transformation, if the mouse hovers over an element in the rewriting rule. Thus, we can check which elements were matched by which item in the LHS of the rule, and which new elements were created after the application of the rule. Using several animators to provide a solution, we can clearly separate orthogonal aspects of the problem space.

State machine models

Figure 5 presents the internals of the *SIM_MatchHighlighter* animator. Recall that this animator highlights those elements of the host model which are matched by the LHS element under the mouse cursor, and the elements of the output model that belong to the RHS element under the cursor. The default state of the animator is the *Matching* state. In case of a new match (*PreApplyCurrentMatch* event is received), the state machine stores the match in its *lastMatch* local variable, and resets the *lastResult* variable storing the newly created elements of the last rewriting. The appropriate guard condition is:

```
PortGT.PeekIsOfType<EH_GT.PreApplyCurrentMatch>() &&
PortGT.PeekAs<EH_GT.PreApplyCurrentMatch>().Match!= null
```

The corresponding action expression is:

```
lastMatch =
PortGT.PeekAs<EH_GT.PreApplyCurrentMatch>().Match;
lastResult = null;
```

The *PostApplyCurrentMatch* transition triggers an event with the same name, and stores the set of newly created elements in the *lastResult* local variable.

If the mouse hovers over a model item, a *MouseEnter* event is fired by the UI event handler. We have to filter it only for the nodes in the rewriting rules with the following guard condition:

```
PortPeripherals.PeekIsOfType<EHUI.EventMouseEnter>() &&
PortPeripherals.PeekAs<EHUI.EventMouseEnter>().View.
Model.AgsiMetaID.Equals(META_NODE);
```

The action expression which fires a *HighLight* event through the *PortViews* port for matched or created elements is listed below:

```
List<Node> match;
Node ruleNode =
(Node)PortPeripherals.PeekAs<EHUI.EventMouseEnter>().
   View.Model.AgsiItem;
if (lastMatch.TryGetValue(ruleNode, out match) ||
lastResult != null && lastResult.TryGetValue(ruleNode,
out match))
  foreach (Node n in match)
    Fire ( new EHUI.EventHighlight(this) { Element = n,
Color = Colors.Green }, PortViews);
```

A more complex scenario is implemented by the *SIM_GT* animator. It is responsible for (i) animating the control flow model, including detecting breakpoints and performing jumps, (ii) initiating the execution of rewriting rules, (iii) visualizing the changes of the output model. The internal structure of the animator is depicted in Figure 6. States and transitions in block (1) are used to initialize the transformation, to open the host and create the output model and to obtain a reference to the opened diagrams. The *Executing* state can be considered as a default state of the animation, the processing of the individual elements of the control flow model are initiated and finished in this state. Blocks (3), (4), (5) and (6), (7) are similar in the sense that they are responsible for processing and highlighting the elements of the control flow, namely the start node, edges, rule nodes, decisions and the stop node. Block (4) is entered after receiving a *Pre-*

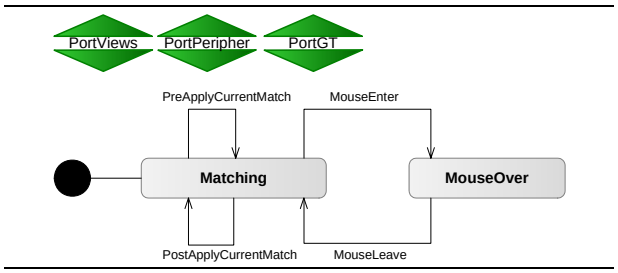


Figure 5. *SIM_MatchHighlighter* animator

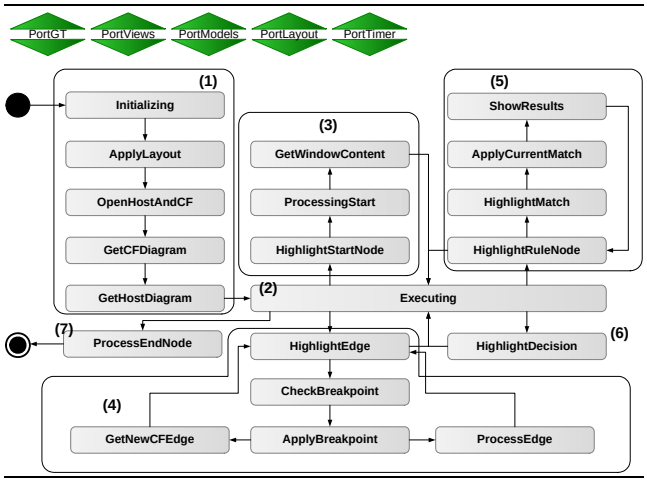


Figure 6. *SIM_GT* animator

NextCFEdge event. At this point we have the possibility to parametrize the *ProcessNextCFEdge* event with another edge, and step to an arbitrary point in the control flow model. This parameter can be set after testing whether an Alt+click event is received (Alt+click on an edge is used to jump to the edge). Consequently the guard expression before the *GetNexCFEdge* state is

```
PortPeripherals.PeekIsOfType<EHUI.EventMouseClicked>()
&& PortPeripherals.PeekAs<EHUI.EventMouseClicked>().
ModifierKeys == ModifierKeys.Alt && PortPeripherals.
PeekAs<EHUI.EventMouseClicked>().View.Model.AgsiMetaID.
Equals(META_EDGE)
```

In case of a successful evaluation of the guard condition, the processing of the control flow jumps to the new edge, otherwise the *ProcessEdge* will be the following state, and the execution continues in a normal way.

Due to the interpreted feature of the graph rewriting engine of VMTS, one can freely edit the control flow and also the host and output models during the debug process as well.

3. Related work

AToM^3 [9] is a general purpose metamodeling environment with simulation and model animation features. AToM^3 supports graph rewriting-based model transformations with a graphical editor for the definition of the rules; furthermore, the interactive debugging of transformations is also possible. The processed model can be animated according to the operations of the model transformation. The transformation can be executed in continuous mode or step-by-step. Compared to VMTS, AToM^3 does not support breakpoints or direct jumps between rewriting rules. Checking the result of a successful match is not possible either.

Graph Rewrite And Transformation (GReAT) [10] is a visual language and toolset to define and execute graph rewriting-based model transformations. GReAT has an approach similar to that of used in VMTS in the sense of graphical rule definition and the sequencing of the rules with a control-flow language. GReAT provides advanced debugging features including step-by-step execution of rules, and the application of breakpoints. The results of successful matches and the results of the rewriting rules are also logged in detail. However, inspecting the operations of the transformations in a visual way is not supported, although one can trace the transformation with the help of a textual interface.

The Attributed Graph Grammar System (AGG) [11] is an environment for developing graph rewriting based transformations. One can follow the execution of a transformation in AGG visually, including the applied rewriting rule and the host graph. The manual definition of matches is also supported by the environment. A transformation can run continuously or step-by-step, however, the process cannot be paused by predefined breakpoints in the rule-application sequence.

MetaEdit+ [12] is a general purpose metamodeling tool. It supports model animation through its Web Service API. Model elements in MetaEdit+ can be animated by inserting API calls into the code generated from the model, or by modifying the code generator to automatically insert these calls. If the attributes of a model element are changed, its visualization is automatically updated. The update mechanism can be influenced with constraints written in a proprietary textual script language of MetaEdit+. The modification of model attributes in VMTS also results in the automatic update of the presentation with the help of data binding. Applying converters to the data binding we can perform an arbitrary transformation on the presented data, this is a similar approach to constraints in MetaEdit+. Compared to VMTS, MetaEdit+ does not provide a graphical notation to define animation or for the integration of external components.

As of writing we are not aware of other visually modeled graph rewriting-based model transformation debuggers. Related work enumerated above provides hard-coded solutions for tracing and debugging transformations.

4. Conclusion

We have presented a visual debugger solution for model processors. The debugger is defined with the help of visual modeling techniques. Building on the VMTS Animation Framework, we could easily connect the animation of the user interface with the model transformation engine.

We have modeled the problem area on three levels. (i) The event handler model is used to wrap the model transformation engine with an event based interface. (ii) The high-level animation model connects event handlers with animators defining orthogonal aspects of the problem. (iii) The state machine models integrate the messages of the user interface and the transformation framework. They decompose the events of the transformation engine to a set of UI events (e.g. opening several diagrams after processing the start node), and also integrate messages of the event handlers into one or several new events (e.g. sending Event*HighLight* events after receiving timer *PreApplyCurrentMatch* and *MouseOver* events). The skeleton of the event handler implementation is generated based on the event handler model; the animation model and the low-level state machines are used generate the executable binaries implementing the debugger.

Future work includes the extension of breakpoints and jumps on further elements in addition to edges, and the improvement of breakpoints to stop the execution only if a predefined condition is satisfied. We would also like to provide a built-in OCL interpreter to evaluate OCL expressions on the transformed models at runtime. Similarly, we would also like to support the modification of causalities, especially the changing of their Imperative OCL code at runtime.

Acknowledgement

This paper was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. The found of 'Mobile Innovation Centre' has supported, in part, the activities described in this paper.

References

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

[2] Mészáros, T., Mezei, G. and Levendovszky, T., A Flexible, Declarative Presentation Framework For Domain-Specific Modeling, Proceedings of the 9th International Working Conference on Advanced Visual Interfaces, 2008, Naples

[3] Mészáros T., Mezei G., Charaf H., Engineering the Dynamic Behavior of Metamodeled Languages, Submitted to Simulation: Transactions of the Society for Modeling and Simulation International, Special Issue: Multi-Paradigm Modeling: Concepts and Tools, 2008

[4] Zeigler B. P., Praehofer H., and Kim T. G., Theory of Modeling and Simulation, Second Edition, Academic Press, 2000.

[5] Ehrig, H. et al., Fundamentals of Algebraic Graph Transformation, Springer, 2006.

[6] Lengyel, L. et al., Control Flow Support in Metamodel-Based Model Transformation Frameworks, Proceedings of the EUROCON 2005 IEEE International Conference on "Computer as a tool", Belgrade, 2005, pp. 595-598

[7] OMG Object Constraint Language 2.0 Specification, http://www.omg.org/docs/ptc/05-06-06.pdf, last visited on 2008.07.16

[8] OMG MOF QVT Final Adopted Specification, http://www.omg.org/docs/ptc/05-11-01.pdf, last visited on 2008.07.16

[9] de Lara, J. and Vangheluwe, H., AToM^3 : A Tool for Multi-Formalism Modelling and Meta-modeling. In ETAPS/FASE'02, LNCS 2306, pp. 174-188. Springer-Verlag

[10] Balasubramanian, D. et al., The Graph Rewriting And Transformation Language : GReAT, Proceedings of the Third International Workshop on Graph Based Tools, 2006, Natal

[11] Taentzer G., AGG: A Tool Environment for Algebraic Graph Transformation, LNCS 1779, pp. 333-341, Springer, 2000

[12] Tolvanen, J-P., MetaEdit+: integrated modeling and metamodeling environment for domain-specific languages, In Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, 2006, Portland

A Domain Specific Design Tool for Spacecraft System Behavior

Sravanthi Venigalla, Brandon Eames
Electrical & Computer Engineering
Utah State University, USA
{sravanthi.venigalla@aggiemail|beames@engineering}.usu.edu

Allan McInnes
Electrical & Computer Engineering
University of Canterbury, New Zealand
allan.mcinnes@canterbury.ac.nz

Abstract

Specification of spacecraft subsystem interactions is typically carried out using informal diagrams and descriptions that can obscure subtle ambiguities and inconsistencies. As a result, problems in the way subsystems are designed to interact may remain undetected until the integration and test phase, when the cost of change is high. Our Behavioral Analysis of Spacecraft Systems (BASS) modeling tool provides a structured way to define spacecraft subsystem interfaces and interactions, and access to an underlying formal model of interaction that allows the specified interactions to be rigorously analyzed. The enforced consistency of the diagrams produced by our tool and the analytical power of the underlying formal model increases a developer's ability to discover and correct system design errors early in the development process.

Categories and Subject Descriptors D.2.2 [**Software Engineering**]: Design Tools and Techniques – computer aided software engineering; D.3.2 [**Programming Languages**]: Language Classifications – specialized application languages, D.3.1 [**Programming Languages**]: Formal Definitions and Theory – syntax, semantics.

General Terms Design, Languages, Verification.

Keywords Domain-Specific Language; Formal Verification; Behavior; Spacecraft System Design

1. Introduction

Spacecraft systems design is the domain of systems engineers, who are responsible for the high level design of not only computer software and hardware, but the physical structure, thermal properties, electrical wiring and harnessing, data communications, orbit management, and spacecraft control. A primary challenge in system-level spacecraft design is ensuring that the spacecraft subsystems interact correctly to allow the spacecraft to achieve its mission. For example, when a ground station commands the spacecraft to begin taking science data from an onboard instrument, that command may trigger a sequence of actions in which the Command & Data Handling (CDH) subsystem instructs the Attitude Determination & Control System (ADCS) to reorient the spacecraft to point at an object of scientific interest, commands the Power subsystem to supply power to the Payload, commands the Payload to begin data sampling, and then collects and stores the sampled science data for a later downlink.

Early in the design process, systems engineers typically develop a simple, high-level system block diagram that shows the partitioning of the system into subsystems, and the connectivity between those subsystems. Separate documentation defines the behavior of each spacecraft subsystem using simple state diagrams, tables, and textual descriptions of how the subsystems respond to internal and external events. Spacecraft subsystem design proceeds under the assumption that the combined behaviors specified for the subsystems will lead to the desired system behavior, and that the subsystems can each be designed in relative isolation as long as each subsystem adheres to the interface and behavior specified for it. However, there is a lack of tool support for capturing spacecraft system behavior specifications, and the mixture of informal notations currently used by systems engineers cannot be readily analyzed, making it difficult to check assumptions about the emergent system behavior until substantial resources have been expended on subsystem design.

In this paper, we introduce BASS (Behavior Analysis for Spacecraft Systems), a model based design tool that supports the modeling and verification of system-level spacecraft behavior, through the analysis of a composition of subsystem behavior models. We focus on the BASS domain-specific visual modeling language (Section 3), which is built upon the Generic Modeling Environment (GME) [1],[2]. This language provides spacecraft-specific modeling constructs that permit both system connectivity and subsystem behavior descriptions to be captured in a single hierarchical model, and provides a starting point from which to gather user feedback on domain-specific modeling needs. A key benefit of providing a language and tool support for describing spacecraft system behavior is the ability to automatically map visual models to an underlying mathematical semantics that permits rigorous analysis. Our semantic model is based on the concept of concurrency and process interaction, and is codified using the CSP (Communicating Sequential Processes) process algebra [3]. As part of the BASS project, we have developed a model interpreter tool, which is responsible for translating the visual models of spacecraft behavior into a CSP model that can be verified against a higher-level specification, or checked against user-specified assertions or constraints. In Section 4 we briefly describe the model interpreter, and the overall toolflow within which spacecraft system models can be constructed and verified.

2. Background

Spacecraft Systems

As with most complex systems, spacecraft designs are usually partitioned into functionally distinct subsystems. Although the exact names and functionality of the subsystems vary from organization to organization, unmanned spacecraft are typically divided into some variation on the following subsystems:

- **ADCS**-- Attitude Determination & Control, responsible for determining the direction the spacecraft is pointing, and for adjusting that direction as needed
- **CDH**-- Command & Data Handling, consisting of the main spacecraft computer system. CDH is responsible for manag-

ing spacecraft interactions with the ground station, as well as collecting, logging and transmitting data

- **Communications**—Transmission and reception of commands and data
- **Power**-- Consisting of the power generation (ex. solar panels), storage (batteries) and distribution (wiring) facilities
- **Payload**-- Offering a mission-dependent subsystem, typically involving some science based instrument or communication device
- **Propulsion**-- The facilities to physically alter the spacecraft velocity and/or position
- **Structures & Mechanisms**-- Physical support for the other subsystems, and deployment of booms, antennas, and solar arrays
- **Thermal Control**-- Regulation of the spacecraft thermal state

Both the behavior of individual subsystems and the interfaces between the subsystems are extremely mission-dependent. Some spacecraft omit subsystems that are unnecessary to their particular mission.

The Generic Modeling Environment

GME is a tool developed at Vanderbilt University for supporting the development and use of domain specific visual modeling languages. Each modeling language dictates a set of rules about the types of parts available, containment relationships and inter-object relations such as connectivity. These rules are codified in a configuration file called a *paradigm*. Once a particular paradigm has been loaded into GME, GME supports the editing of models according to that paradigm. GME supports the partitioning of a system into views called *aspects*, facilitating the separation of concerns.

GME is packaged with a modeling paradigm, called MetaGME, which supports the creation of *metamodels*, or models of modeling languages. With MetaGME, users can define a new language which conforms to a particular engineering domain. A translator tool produces a paradigm from a valid metamodel. The metamodeling language is an extension of UML class diagrams, and offers the flexibility to integrate concepts such as hierarchy, inter-object relationships, object attributes and referencing into a modeling language.

GME also offers multiple APIs or interfaces for creating translator tools called *interpreters*. GME allows an interpreter to access the information captured by the user when drawing models. Interpreters apply semantic translations, performing such tasks as code generation, model-to-model transformations or model analysis. Multiple language bindings, including C++ and Java are supported.

Communicating Sequential Processes

Communicating Sequential Processes (CSP) is a mathematical theory of concurrency and interaction, in which interacting processes are modeled as event-transition systems that synchronize on shared events. The fundamental objects from which CSP process models are built are *events*, which are abstract symbolic representations of interactions. For example, a model of a financial transaction might consist of events that represent placing an order, acknowledgement of the order, payment, providing change, and handing over the purchased goods. Simple processes are built by defining sequences of events, separated by the prefix operator →, e.g.

```
SellEspresso =
  espresso_order → order_cost!$3 →
  receive_payment?p → make_change!(p - $3)
  → give_espresso → SKIP
```

CSP also provides a variety of operators for defining behaviors such as alternative actions (**SellEspresso [] SellLatte**), nondeterministic outcomes (**(espresso_order → Transaction) |~| (out_of_coffee → CloseStore)**) sequences of processes (**LoneBarista = SellEspresso; SellEspresso;** …), parallel execution of processes (**TwoBaristas = LoneBarista ||| LoneBarista**), and interfaces between processes (**Customer [|OrderEvents|] TwoBaristas**).

CSP supports a rich theory of process equivalences and refinements. Industrial strength tools such as FDR2 [4] can be used to rapidly check process models for properties such as deadlock, livelock, or refinement of a more abstract specification process. FDR2 has been in use for over a decade, and has been applied to a variety of applications across several domains, from industrial applications [5] and defense applications [6], to hardware design verification [7].

3. Modeling Spacecraft Behavior

Spacecraft systems designers have traditionally examined behavior only informally. Often, diagrams are used, but only for documentation. Consequently, there is no widely adopted standard for graphically representing spacecraft behavior. The BASS modeling paradigm, presented here, represents a starting point for the development of a design tool. The design of the graphical syntax was influenced both by currently employed informal notations, as well as by constructs developed by McInnes [8] for modeling spacecraft behavior using CSP. We envision an iterative development model for BASS, using feedback from spacecraft systems designers to improve the language.

System-Level Modeling

The subsystems and their connections to one another are

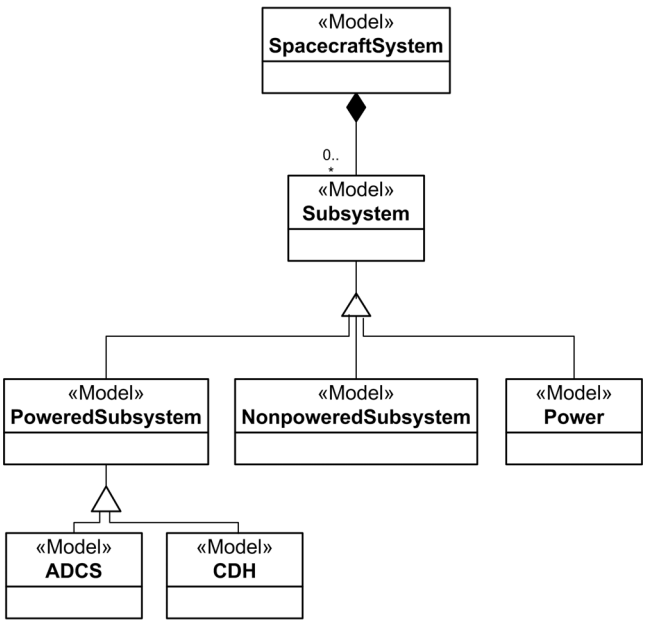


Figure 1. SpacecraftSystem and three types of Subsystems: Power, ADCS and CDH

captured in a top-level *SpacecraftSystem* model (Figure 1) that corresponds to a typical spacecraft system block diagram. We classify subsystems based on power consumption: some subsystems are powered, other subsystems are not powered (ex. structure). As the provider of power, the Power subsystem is in a category by itself.

The *SpacecraftSystem* offers two aspects, separating the views of power connectivity from data connectivity in the system. The parts available in the *PowerAspect* view are shown in Figure 2. The Power subsystem may contain several *PowerPorts*. Each *PowerPort* is capable of delivering power to another subsystem. The topology of the *PowerPorts* also models the structural connectivity of the power distribution network (star topology, single power bus, multiple power busses, etc). The *PowerConnection* connects the *SubSysPowerIf* to the *PowerPort* of the Power subsystem, representing the connection of the subsystem to the power network.

Data communication between subsystems occurs in multiple ways, as depicted in Figure 3. The primary vehicle for data communication is a *SystemBus*. Spacecraft may have multiple, independent busses, redundant busses, or a single bus, depending on the mission and resource availability. The bus carries multiple types of information. First, commands can be issued by the CDH subsystem to other subsystems. The set of commands accepted by a subsystem is captured as a *CommandSet*. Commands issued by the CDH are carried by a *SystemBus*. Command transmission is associated with a particular bus instance via the *CommandInterface* connection. The means by which the user models how the CDH selects which commands to send will be shown below.

The second type of information carried by the bus is spacecraft state information, which typically includes data indicating the current health of the spacecraft (ex. current temperature, position data, power level, etc.). State information can be used by the CDH to make operational decisions, and is also often stored for later downlink to the ground station. State information may be sent over a *SystemBus* as discrete responses to individual requests, or may be transmitted as a stream of telemetry data. Streams of information are represented using the *TelDataStream* construct (not shown), to which the *TelDataStreamRef* refers. The

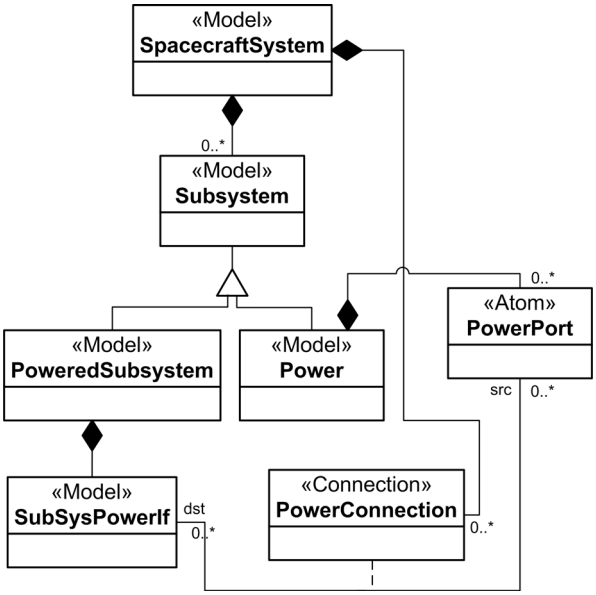


Figure 2. PowerAspect view of SpacecraftSystem

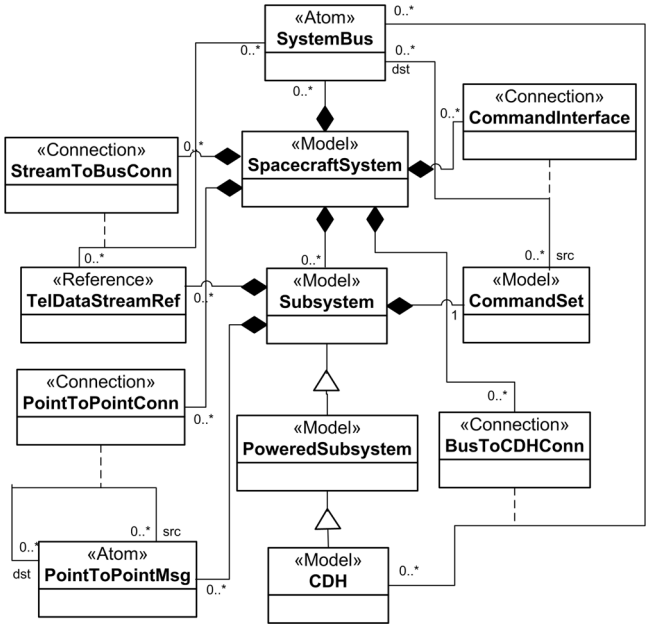


Figure 3. *SpacecraftSystem DataCommAspect*, showing data connectivity between the subsystems and CDH

StreamToBusConn allows the user to associate a stream with a particular *SystemBus*.

PointToPointMsgs are discrete messages sent from one subsystem to another. Physically, these messages are routed on dedicated wire connections between subsystems, modeled with the *PointToPointConn*. These messages are used to convey discrete packets of information which are not streamed, ex. an image captured by a science instrument to be recorded by CDH.

Modeling Subsystems

The system-level diagram specifies what subsystems are present in the spacecraft, and specifies paths for their interaction. The

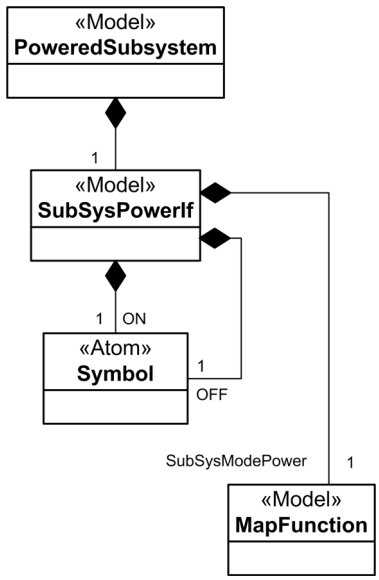


Figure 4. Power interface used by all powered subsystems

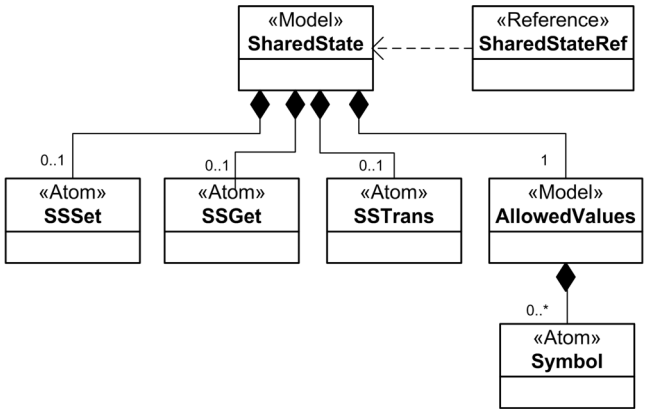


Figure 5. *SharedState* object, representing a shared variable

actual behavior of those subsystems is individually captured within each subsystem model.

Prior to discussing individual subsystems, we discuss some common constructs reused across multiple subsystems. We then discuss three types of subsystems: the Power, CDH and ADCS. Our discussion only summarizes the modeling facilities BASS offers to model subsystem behavior, with many low-level modeling details omitted.

Common Subsystem Constructs

Powered subsystems must interact with the Power subsystem. Each powered subsystem must specify a *SubSysPowerIf*, as shown in Figure 2. Figure 4 shows the internals of the *SubSysPowerIf*. The minimal power interface consists of two *Symbol* objects, one representing power to the subsystem being switched on, and the other representing power being switched off. These symbols are translated into CSP events by the model interpreter, and can be referred to in other parts of the system model. In addition to simple on and off states, some subsystems consume varying amounts of power depending on the mode they are in. The *MapFunction* allows the user to model this behavior, capturing a mapping between each mode of the subsystem and a corresponding change in power consumption.

Interactions between subsystems that are caused by dependencies on physical states are modeled in BASS using a *SharedState* object (Figure 5). *SharedState* objects have a well-defined type alphabet, as well as a well defined interface (*Set*, *Get* and *Trans* ports) for accessing the state.

As mentioned above, telemetry streams represent continuous flows of state information transmitted from one subsystem to another. During early phases of spacecraft design, specific values associated with streamed data are usually less important than the qualitative ranges of values that will trigger specific actions.

Furthermore, explicit enumeration of every possible value the streaming data can take on would inevitably produce a state explosion during model checking. Therefore, we restrict our stream model to qualitative transitions in the value of the state information the stream carries (see [8] for further details).

Power Subsystem

The Power subsystem is responsible for producing, storing and delivering power for the spacecraft. The most common kind of spacecraft uses solar arrays to generate power, and batteries to store power. Our current Power subsystem model focuses on solar-battery systems, and in particular attempts to address the fact that the amount of power that can sustainably be delivered by the Power subsystem can be a function of the attitude of the spacecraft (the attitude determines the angle at which on-board solar panels face the sun; angles approaching 90° result in higher power generation). The Power model (Figure 6) has two attributes, defining the minimum and maximum power generation capability of the spacecraft. The *PowerPorts* model the power interface to the outside world. The *MapFunction*, contained in the role of *AttitudeSpecificAvailablePower* is responsible for defining a mapping between spacecraft attitude and the power level available when the spacecraft is operating in that attitude. The definition of the *MapFunction* is omitted, but allows the user to associate a *Symbol* object, representing an attitude, with another *Symbol* object, representing a power level. Note that the Power subsystem also inherits containment of a *CommandSet* and *TelDataStreamRef* from the Subsystem class as shown in Figure 3. Hence the power subsystem can receive commands from CDH, and can stream health/status information back to CDH.

CDH Subsystem

The Command and Data Handling subsystem is responsible for coordinating the various subsystems onboard, logging state information, and interacting with the ground station. We consider separately two portions of the CDH subsystem: command and control, and data handling. Command and control consists of receiving commands from a ground station and dispatching them appropriately. A command received from the ground station may involve sending a single command to one subsystem, but frequently involves issuing a sequence of commands, where one command must complete before the next is issued. Figure 7 illustrates how commands are modeled. A *SimpleCommand* may be parameterized with a set of Symbols. A *CommandSequence* consists of multiple *Commands*, whether they be *SimpleCommands* or other *CommandSequences*. The *CommandSequencing* connection imposes a linear order on the *Commands* contained in the command sequence. *CmdRef* is a Reference to another command, for example a command belonging to a different subsystem. *SymbolMappingConn* connections can be used to bind the parameters of one command to the parameters of the following command in a command sequence.

From a modeling perspective, the specification of how a command is handled when it is received by the CDH involves defining a mapping from a command in the CDH command set onto either a *CommandSequence* or a *SimpleCommand*. The target command or sequence may be drawn from either the CDH command set, or from the command set of a different subsystem. Figure 8 depicts how command dispatching is modeled in BASS. *CDHCmdDispatch* consists of sets of <*Trigger, Target*> pairs.

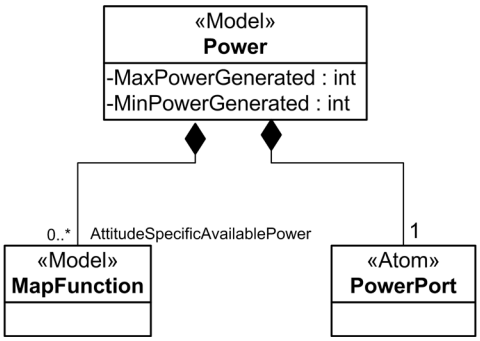


Figure 6. Power subsystem

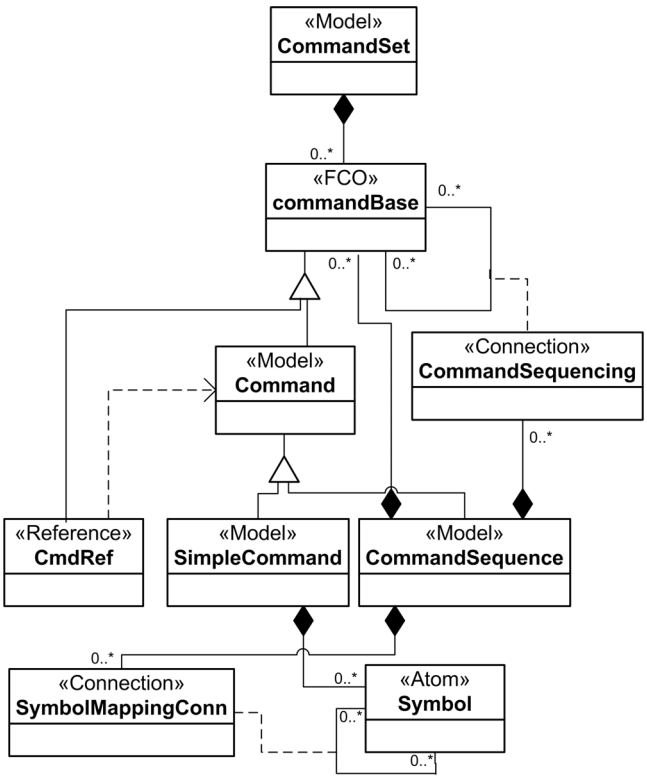


Figure 7. Spacecraft Commands

The *Trigger* is a reference to a command from the CDH command set which represents a command received from a ground station. The *Target* represents the result of the command dispatch, and can either be a reference to a command, or a *Symbol*. The *Symbol* is used to model the raising of an event, or the communication of a scalar flag to some subsystem. For example, in a command to the Power subsystem to turn on the power to ADCS the *Symbol* would be the *On Symbol* contained in the ADCS *SubSysPowerIf*. The *SymbolMappingConn* is used to indicate a mapping between the parameters of the *Trigger* to the parameters of the *Target*.

Attitude Determination and Control

ADCS is responsible for determining and maintaining spacecraft attitude, subject to commands issued by the CDH subsystem. Since we are concerned with system level behavior as a function of subsystem behavior, we abstract from the continuous dynamics control laws (which may be undefined during early design phases), and instead model the ADCS as a supervisory mode transition system. We assume that the ADCS includes one or more controllers that are capable of adjusting the spacecraft to attain the nominal attitude associated with a given ADCS mode when that mode is entered. Later design and analysis work by a control systems expert would be required to ensure that the ADCS does indeed meet this assumption. However, for the sake of high level behavioral analysis, the assumption allows us to determine whether the attitude changes resulting from a transition in ADCS mode cause, for example, undesirable changes in the available spacecraft power.

The *ADCSModeSystem* (Figure 9) is composed of *ADCSMode* objects, which model the ADCS modes, and *Symbol* objects, which model the rules for transitions between modes. The mode

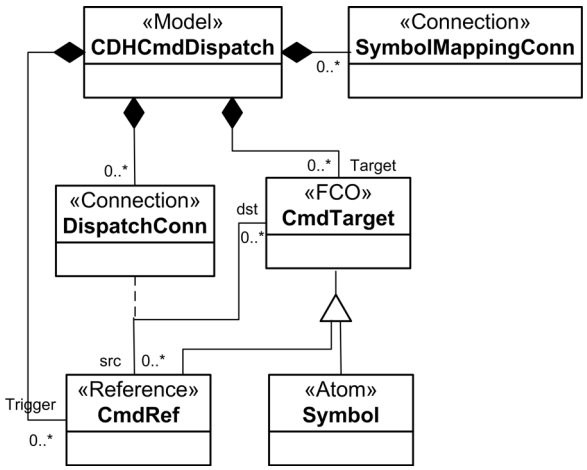


Figure 8. CDH Command Dispatch

transition *Symbols* may include *Symbols* present in the ADCS *CommandSet*, allowing receipt of a *SimpleCommand* to trigger an *ADCSModeSystem* transition. The *ADCSModeSystem* also contains an *AttitudeSet* object, which is a set of *Symbols* representing the nominal attitudes attainable by the spacecraft. These attitudes are associated with modes through the *AttToModeMap* connection. Each *Mode* must be associated with an attitude, but *Modes* may share attitudes. Also associated with each mode is a *ModeSpecificFn*, which represents the actions to be taken while in a particular mode. Such actions could include interacting with *SharedStates*, sending signals, or modifying telemetry streams.

Figure 10 depicts an example *ADCSModeSystem* containing three modes, *Safehold*, *Sci_Active* and *Sci_Standby*. Solid lines connecting modes to symbols ("sym" objects) model transitions. The symbols involved in transitions correspond to commands

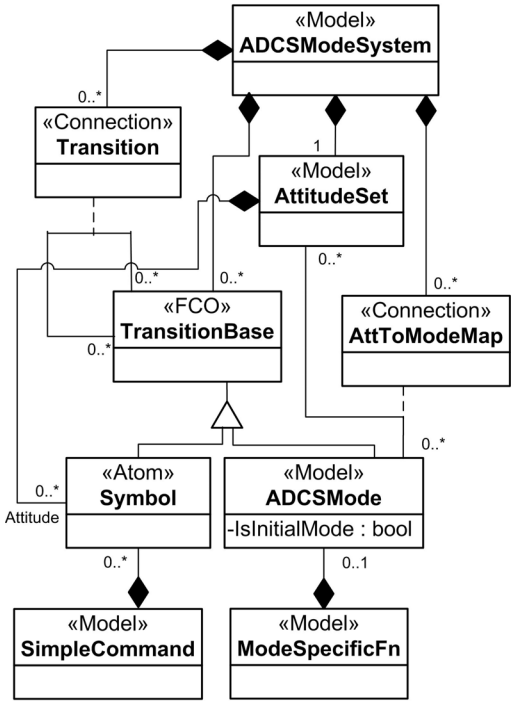


Figure 9. ADCS Mode System

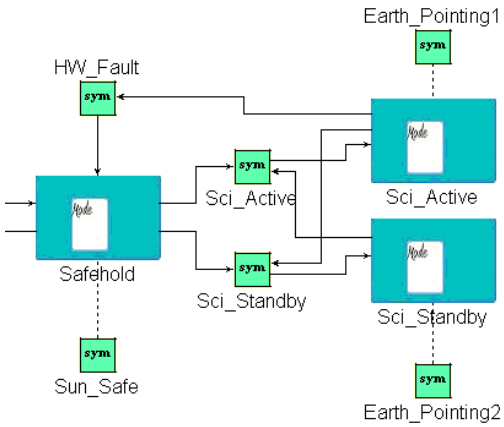


Figure 10. Portion of an ADCSModeSystem

received from the CDH, or to some other event (ex. *HW_Fault*) that can cause mode transitions. A dashed line connecting a symbol to a mode shows the mapping between attitude and mode.

4. BASS Toolflow

BASS offers the ability to model a spacecraft at the system level using the modeling language described in detail above. The modeling language is only one part of the BASS tool, as depicted in Figure 11. Development begins with the capture of system behavior models using GME and BASSML. An example system level diagram is shown in Figure 12. This example has only three subsystems – Power, ADCS, and CDH, communicating over a single system bus. The CDH port within the CDH model is actually a *CDHCmdDispatch* model, and contains the rules for how commands received from the ground station are dispatched to other subsystems via the System Bus. The Com ports of Power and ADCS are of type *CommandSet*, and contain command definitions for their respective subsystems. The connections between ports named Att and the *SystemBus* model the communication of the current attitude via a telemetry data stream from the ADCS to CDH. The models shown in Figure 12 are further refined into other diagrams which are omitted for brevity.

Once the system is captured in the BASS Modeling Language (BASSML), the Interpreter is applied to translate the model into

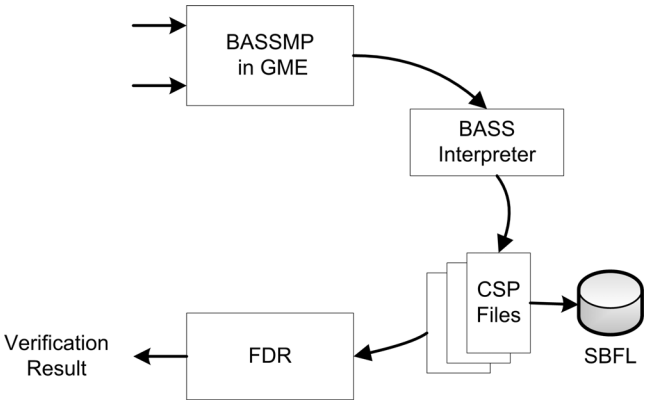


Figure 11. BASS Tool-flow

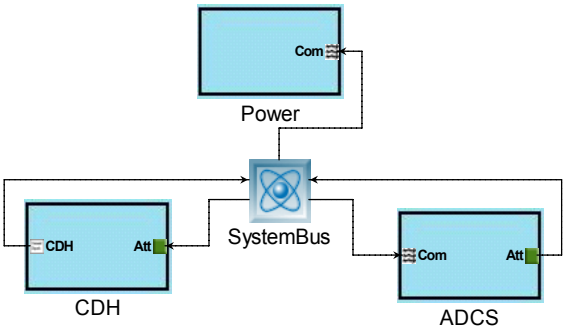


Figure 12. CDH Aspect of an example spacecraft model

machine-readable CSP. For each data communication path in the model, the interpreter produces a channel. Each *Symbol* defined in the model corresponds to an event which can be sent over a channel. Subsystem behavior is encoded as a set of processes, which interact using the generated channels. Our underlying process-based semantic model for BASS is described in detail in [8], which also describes a library of CSP processes for modeling spacecraft behavior (the "Spacecraft Behavior Framework Library", or SBFL) that is heavily used by BASS. The CSP model generated by the interpreter can be sent to the FDR2 model-checker for verification of specific behavioral properties (e.g. the spacecraft never reaches a deadlock state), as well as confirmation that nominal mission scenarios are feasible or that the system design implements a higher-level specification of system behavior (e.g. a functional flow block diagram). Analyses with FDR2 can be used to detect unanticipated interactions between subsystems that lead to errors such as activating a payload while the spacecraft is in an attitude that could damage the instrument, or improperly transitioning into a mode that require more power than is currently available. Such errors are often subtle, and can easily be overlooked during a cursory visual review of a model.

BASS can be used by spacecraft systems engineers throughout the entire system lifecycle, but is primarily intended to support specification and analysis in the preliminary design phase - what NASA calls "Phase B" [9]. Engineers at the Space Dynamics Laboratory and Air Force Research Laboratory have both expressed some interest in using a tool like BASS. However as yet, BASS has not been used outside of the laboratory, and we intend to refine the tool further via experiments with specification and analysis of student satellites such as USUsat1, USUsat2, and Toroid before releasing it to a wider audience.

Although BASS itself has not yet seen extensive use, our initial experiments with analyses of example specifications developed using the underlying CSP semantic model have shown that these analyses can be useful for uncovering several different kinds of errors, including

- **Interaction design errors:** for example, a mission-ending power-up sequencing error that escaped manual review (and indeed that the design had been specifically created to avoid);
- **System specification errors:** for example, an incompatibility between the subsystem interaction model and a higher-level system behavior specification (an FFBD) which exposed omissions in the higher-level specification;
- **Operations planning errors:** for example, a faulty commanding scenario that failed to place the spacecraft into the correct attitude for data gathering.

5. Related Work

Applying formal methods to spacecraft analysis is certainly not a new topic, although the focus of previous efforts has largely been on individual elements of software or hardware rather than on system-level interactions, and none have involved development of domain-specific languages.

NASA has carried out several experiments with formal methods. An analysis of flight software based on model extraction directly from source code into the SPIN model checker has been examined [10], and exposed design-level problems in the legacy software of the Deep Space One mission. Easterbrook *et al*. [11] successfully applied the PVS theorem-prover to check software requirements for consistency, and for safety and liveness properties. CSP has been evaluated and proposed for use as a specification language for use in the NASA ANTS mission architecture [12], and in the Formal Approaches to Swarm Technology project for specifying and verifying SWARM based missions [13].

Some tools offer a graphical interface to support formal verification. Hilderink developed a graphical modeling tool that has constructs for representing system behavior, and generates machine-readable CSP [14]. The generated CSP can be model-checked in FDR. However, the language constructs are generic and CSP-specific, rather than being designed for an application domain such as spacecraft design.

Specification Description Language (SDL) is another graphical specification language which uses formal methods [15]. SDL is based on Finite State Machines (FSM) and can be used to describe system behavior. However, it is more widely used for telecommunication systems and to our knowledge, has not been applied widely to spacecraft. However, it has been applied to the validation of fault tolerance in the design of autonomous spacecraft, examining in particular the Data Management System[16].

6. Conclusions and Future Work

Spacecraft system design is difficult, and can lead to expensive, even catastrophic consequences when subtle design flaws are not caught early in the design process. In this paper, we present BASS, a prototype modeling tool for spacecraft systems. BASS utilizes a domain specific language targeting spacecraft designers. BASS integrates a model interpreter, capable of translating the captured spacecraft design models into machine readable CSP, which can be formally verified using the FDR2 model checker.

As part of our efforts to further refine BASS, we intend to examine and incorporate lessons learned from similar initiatives in other domains, such as the AUTOSAR-based modeling in the automotive domain [17] and MIMAD in the avionics domain [18] We will also explore closer integration of BASS with tools for spacecraft requirements capture. A prototype behavioral requirements capture tool called SDW [19], which we developed previously, is a particularly good candidate for integration efforts, since like BASS it relies on CSP for its semantic model.

References

[1] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing domain-specific design environments," Computer, vol. 34(11), pp. 44-+, NOV, 2001.

[2] "GME User's Manual," Vanderbilt University March 2004

[3] A. W. Roscoe, The Theory and Practice of Concurrency. Englewood Cliffs, New Jersey: Prentice Hall, 1998.

[4] P. Gardiner, M. Goldsmith, J. Hulance, D. Jackson, B. Roscoe, B. Scattergoood, and P. Armstrong, "Failures-Divergence Refinement: FDR2 User Manual," Formal Systems (Europe) Ltd. May 2003

[5] B. Buth, J. Peleska, and H. Shi, "Combining methods for livelock analysis of a fault-tolerant system," Proc. 7th International Conference on Algebraic Methodology and Software Technology (AMAST'98), Jan., 1999, pp 124-139.

[6] M. Goldsmith and I. Zakiuddin, "Critical systems validation and verification with CSP and FDR," Proc. International Workshop on Current Trends in Applied Formal Methods (FM-Trends 98), Boppard, Germany, Oct 7-9, 1998, pp 243-250.

[7] G. Barrett, "Model checking in practice: The T9000 Virtual Channel Processor," IEEE Transactions on Software Engineering, vol. 21(2), pp. 69-78, 1995.

[8] A. I. McInnes, "A Formal Approach to Specifying and Verifying Spacecraft Behavior," Ph.D. Dissertation, Utah State University, 2007

[9] R. Shishko and R. G. Chamberain, NASA Systems Engineering Handbook: NASA SP-6105, 1995.

[10] P. R. Gluck and G. J. Holzmann, "Using SPIN model checking for flight software verification," Proc. 2002 IEEE Aerospace Conference, 2002.

[11] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, "Experiences using lightweight formal methods for requirements modeling," IEEE Transactions on Software Engineering, vol. 24(1), pp. 4-14, 1998.

[12] C. Rouff, A. Vanderbilt, W. Truskowski, J. Rash, and M. Hinchey, "Properties of a Formal Method for Prediction of Emergent Behaviors in Swarm-based Systems," Proc. Second International Conference on Software Engineering and Formal Methods (SEFM'04), 2004.

[13] M. Hinchey, J. Rash, and C. Rouff, "Some Verification Issues at NASA Goddard Space Flight Center," Proc. Verified Software: Theories, Tools, Experiments, 2005.

[14] G. H. Hilderink, "Graphical modelling language for specifying concurrency based on CSP," IEE Proceedings: Software, vol. 150(2), pp. 108-120, Apr., 2003.

[15] A. RockStrom and R. Saracco, "SDL-CCITT Specification and Description Language," IEEE Transactions on Communications, vol. COM-30(6), June, 1982.

[16] S. Ayache, E. Conquet, P. Humbert, C. Rodriguez, J. Sifakis, and R. Gerlich, "Formal Methods for the Validiation of Fault Tolerance in Autonomous Spacecraft," Proc. The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96), 1996.

[17] "Specification of Interaction with Behavioral Models," Doc. No. 205, V1.0.5, Feb 14 2008. http://www.autosar.org/download/AUTOSAR_InteractionBehavioralModels.pdf

[18] A. Gamati, C. Brunette, R. Delamare, T. Gautier, and J.-P. Talpin, "A modeling paradigm for integrated modular avionics design," Proc. 32nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA06), Cavtat/Dubrovnik, Croatia, Aug, 2006.

[19] B. Eames, A. I. McInnes, J. E. Crace, and J. M. Graham, "A Model-Based Design Tool for Systems-level Spacecraft Design," Proc. 20th Annual AIAA/USU Conference on Small Satellites, Logan, UT, August, 2006.

Using Integrative Modeling for Advanced Heterogeneous System Simulation

Tapasya Patki Hussain Al-Helal Jacob Gulotta Jason Hansen Jonathan Sprinkle

Department of Electrical and Computer Engineering
University of Arizona, Tucson, AZ 85721-0104
{tpatki,alhelal,jgulotta,jchansen}@Email.Arizona.Edu/sprinkle@ECE.Arizona.Edu

Abstract

This paper is an academic experience report describing the use by researchers at the University of Arizona of a domain-specific language developed by the Institute for Software Integrated Systems (at Vanderbilt University). The domain in question is heterogeneous, distributed simulation of quad-rotor unmanned aerial vehicles (UAVs) as they respond to command and control requests from a human operator. We describe in detail how our individual designs of the controller and guidance laws for the UAV, as well as its rendering and position updates in a 3D environment, its onboard sensors detecting ground features, and the various commands to delegate mission-critical behaviors, all interact using the ISIS-developed modeling language. We then discuss the outlook for this domain (heterogeneous system simulation and integration) for domain-specific languages and models, specifically for unmanned vehicle control and interaction.

Categories and Subject Descriptors D.2.6 [*Programming Environments*]: Integrated Environments

General Terms Domain-specific modeling languages, Heterogenous system simulation

Keywords HLA, RTI, GME, Command and Control (C2)

1. Introduction

Large projects with decentralized development face a critical issue in holistic system simulations. Maintaining a single simulation strategy, which may even include the use of proprietary tools and/or shared network drives, is quite difficult to achieve, and can lead to poor software engineering practices where elements are developed outside the simulation toolchain. These elements must be rewritten or adapted to fit inside the tools used by the project. Such practices are prone to problems that are subtle, such as mismatched models of computation, as well as problems that are widespread, such as software bugs while porting.

Many systems require development and design in proprietary tools (e.g., MATLAB/Simulink for the domain of control systems), and may take advantage of sophisticated models of computation available in such tools. Other portions of the system may depend on logic that is best expressed as a switch statement in C/Java, or may be run as an applet (e.g., human control through a command and control interface). How to integrate these portions of the system with various components written in other languages is best done through middleware, and many standard middlewares exist for such applications. However, for an expert in control, or discrete event simulation, middleware programming can be a treacherous and confusing addition to their own algorithms.

The simulation of these systems built with heterogenous tools, components, models of computation, and operating systems is a nontrivial task that is best tackled by a middleware expert. However, there exists the bootstrapping issue of confirming that all programmers for each component follow a styleguide, or include standard header files with standard object definitions. Enforcing such a styleguide early in the process can often lead to the 'chicken-and-egg' problem where experts cannot start working on their algorithms because they do not have a testing infrastructure, while infrastructure developers cannot develop the middleware because they do not have a set of algorithms to design around.

To address this issue for the specific domain of multi-vehicle command and control (C2), Balogh et al. [2008, Balogh] developed the HLA paradigm. Starting with a suite of tools that could utilize the infrastructure, and with a few examples, we began an experiment to continue advanced implementation of interactions between components, with the intention of integrating the components into an advanced demo that drew from many different simulation, design, and visualization tools. Importantly, we were able to do our component designs and simulations *independently* of the anticipated middleware, infrastructure, and global simulation strategy. Although it was known *a priori* that HLA was the likely candidate, this strategy enabled users to operate without that assumption[1]. Integration of these various components was somewhat trivial, which is a great result for the domain-specific modeling language, as it reduced the complexity of the expert developers significantly.

The scope of this paper does not include motivating the development of this HLA modeling language, nor a detailed description of the HLA middleware used. Readers interested in these details can refer to Balogh et al. [2008, Balogh][2]. In fact, there were many design choices and application domain choices made by the authors of the domain-specific language we use in this paper: we do not justify or motivate their work, but instead present this application example, which shows the tremendous amount of heterogeneous simulation, design, and rendering, which the use of this domain permitted in the period of just three months. For this paper's scope, we are most interested in the following qualities of a modeling language:

- the ability to specify tool-independent data structures;
- the ability to compose data structures with other data structures;
- the ability to synthesize "glue code" between various tools and software architectures;
- the ability to prototype component behaviors without running middleware as part of the test;

[1] In fact, early application domain choices utilized the ICE middleware by ZeroC.

[2] The maturity of the project and the short timeline for this workshop do not permit an in-print citation of the work.

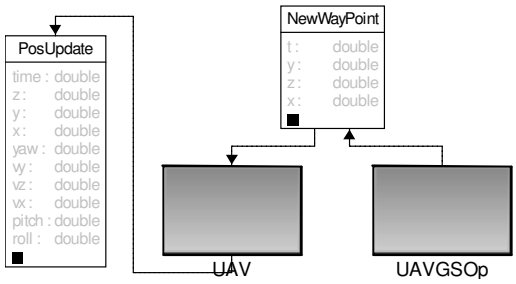


Figure 1. GME Model in the HLA paradigm, which describes an interaction between the `UAV` and the `UAVGSOp`. Note that `New Waypoint` information is passed to the `UAV` from the ground station, and the `UAV` publishes its position for interested parties. The object-oriented structure of those messages are present in the UML class diagram representation.

- the ability to use existing domain-specific tools and environments for design of models, and re-use those models in those tools at runtime; and
- the ability to have a single, unifying modeling language that permits all of the above.

In this paper, we describe our experience with using this domain-specific modeling language, specifically with its advancement of our design and simulation agenda from the perspective of "what would we have to do if we did not have the modeling language to help?" We first describe the tools which were at our disposal for design, simulation, and visualization. We next discuss the various implementations of component functionality. Finally, we describe the integrated demonstration, and how we envision our future work based on the capabilities of this modeling environment.

2. Modeling Language Description

The Generic Modeling Enviroment (GME) [Ledeczi et al. 2001] developed by the Institute for Software Integrated Systems (ISIS) at Vanderbilt University is a configurable modeling environment with a graphical interface. The basis of GME is Model Integrated Computing (MIC) [Ledeczi et al. 2005], which facilitates the creation and evolution of embedded software systems that are domain-specific. MIC gives us an end-to-end view of the entire system and lets us capture an entire class of applications. The Multi-Graph Architecture (MGA) supports *meta-level* processes and *meta-modeling*, and is an executable framework for MIC. A meta-level process is used to specify, define and validate domain-specific environments. These environments are then used to obtain metamodels, that let us specify the various properties of the *domain models*, like the concepts, relationships, compositions and integrity constraints. Metamodels are used to generate *paradigms*, that can be deployed to create applications.

The project required the integration of various heterogeneous tools. Specifically, we needed the ability to use

- MATLAB/Simulink
- OMNeT++ [Varga 2001]
- CPN Tools [Jensen et al. 2007]
- DEVSJAVA [Palaniappan et al. 2006]
- 3D-Viewers like Delta3D and Blender.

To facility communication between these tools, the High Level Architecture (HLA) [IEEE-HLA-1516 2000] was used to handle interoperability and synchronization issues. HLA is an object-based, DoD/IEEE standard that facilitates the building of simulation systems from heterogenous components, permits component reuse, and supports interoperability through publish/subscribe mechanisms. It also encourages distributed and multi-platform computing. An elementary HLA-compliant program is called a *federate*, and a set of interacting federates is referred to as a *federation*. The exchange of data among federates is supported via the Runtime Infrastructure (RTI). The key functions of the RTI involve Federation Management, Data Distribution Management and Time Management. To support these functionalities, various toolkits like the MATLAB HLA Toolbox, Open-HLA, pRTI and Portico are available. Authors used the Portico 8.0 environment to implement the RTI. Portico is a flexible, open-source, cross-platform implementation of the RTI (see `http://www.porticoproject.org/`).

The HLA paradigm, developed by ISIS at Vanderbilt University, was used to create various models in GME, including a particular interaction model shown in Figure 1. The federates corresponding to the `UAV` and the `Ground Station Operator` are components developed in *any* tool that the paradigm supports (in our case, MATLAB/Simulink for `UAV`, and Java for `UAVGSOp`). In this case, the `UAV` publishes updates to its position, but the `UAVGSOp` can send new commands via the `NewWayPoint` message, which will affect the direction in which the `UAV` flies.

These domain models specify the fundamental structure of the interconnection of these components, and the messages they are allowed to send. Model interpreters specify a mapping between this structure and the RTI infrastructure, and various application components, in the following ways. Although their detailed description is out of the scope of this paper, the HLA paradigm has two model interpreters—the `C2WInterpreter` and the `CPNInterpreter`. The first of these interpreters produces the required federate files that are used by the RTI. These files include the names of various components, the data structures expected to be shared, etc. The second, (`CPNInterpreter`), imports information from a CPNTools model to obtain the discrete events expected from that tool, for use in integrating the overall demonstration. These discrete events are then used in the HLA model to specify which events are expected to be received from which components, and consequences of their receipt (i.e., actions). This permits the specification of complex decision-making processes in an existing tool (CPNTools), and the sending of various messages to computational components throughout the model.

The HLA paradigm provides the following abilities:

- layout of interaction models between various components;
- middleware-independent specification of data structures for messaging; and
- specification of runtime parameters for the overall simulation.

Recall from Figure 1 that a ground station interacts with a UAV. The data in that diagram, namely `NewWayPoint`, is fully specified in a larger diagram. The Object and the Interaction Diagram, as modeled in the HLA paradigm, are shown in Figure 2. The notations used are similar to those in standard UML, but the language definition has the additional semantics that their interconnection will denote specific structure in the federate synthesis.

3. Rendering

The aim of a simulation is to accurately replicate a real-life or hypothetical scenario, including its visualization. Tools that permit the high-fidelity design and simulation of a complex vehicle do not always provide an equivalent high-fidelity rendering of that vehicle, so there exists a need to enhance this visualization through external tools. One such tool is Delta3D (`http://www.delta3d.org/`), an open source gaming and simulation engine which provides a 3D

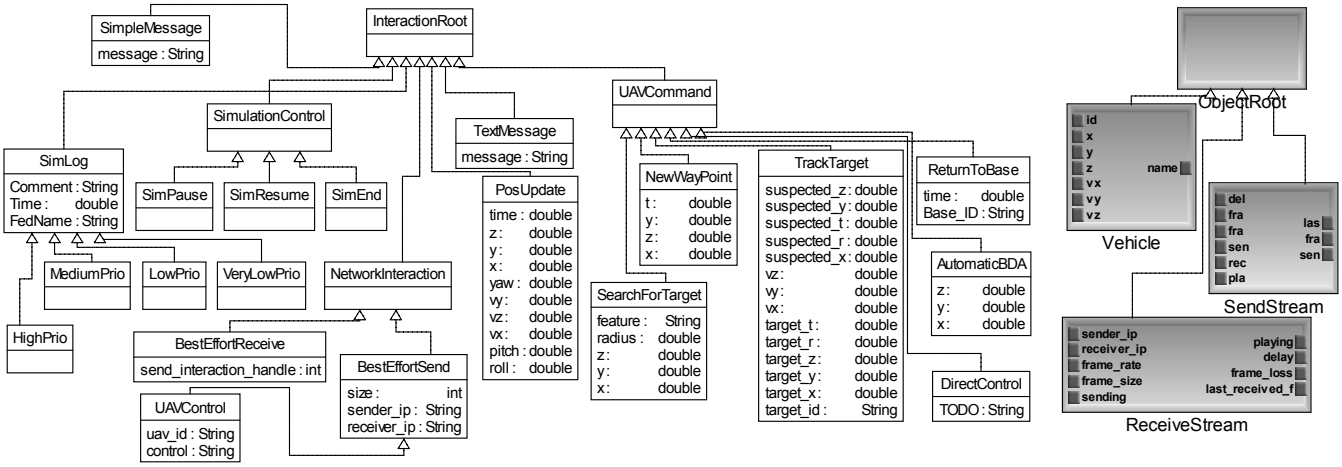


Figure 2. The model of interactions, in the HLA paradigm. Models describing the runtime parameters for simulation, logging, and network interactions are in the left-hand side of the figure. The right hand side concentrates on messages sent back and forth between various components (`UAV`, `Ground Station`, etc.), including the commands used by various components. Finally, the rightmost portion of the figure shows the various objects that are passed by the RTI, including the sending/receiving streams of the network simulator.

visualization of the virtual world as the simulation executes (refer to the manuscript by Balogh et al. [2008, Balogh]).

Delta3D is packed with numerous sprites which can be used to represent a UAV, such as that in Figure 3(a). This is not an accurate representation of the STARMAC [Hoffmann et al. 2007] that is to be simulated, so a more suitable rendered model was created using Blender (`http://www.blender.org/`), an open source 3D content creation suite used to create and render the 3D model seen in Figure 3(b). This is a much more accurate representation of the STARMAC than the included sprite.

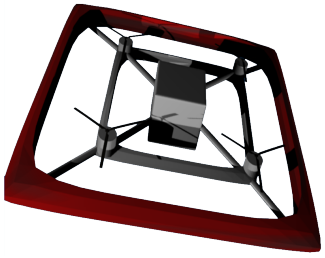
In order to correctly render the UAV, Delta3D requires position (x,y,z), velocity ($\dot{x}$, $\dot{y}$, $\dot{z}$), attitude (ϕ, θ, ψ) and a time stamp ($time$). Since Delta3D can natively interface with HLA, no special GME interpreter is required to generate the glue code for HLA integration. Implementing the HLA Run-Time Infrastructure allows published `PosUpdate` information to be subscribed to by a federate. The Delta3D federate can then be configured to subscribe to `PosUpdate` interactions. HLA allows any source to publish the `PosUpdate` information making development flexible.

Actual image data is not feasible to transfer over the network; network bandwidth causes a large bottleneck. Fortunately, it is possible to send simulated data from Delta3D to look like new data from a sensor. The UAV state data and GPS location corresponding to a target can be published to the RTI. If the UAV is in the `Search Target` State, Delta3D will publish confirmation that the UAV has started its search. The Delta3D federate will constantly publish target position information ($target\_x$, $target\_y$, $target\_z$, $target\_r$, $target\_t$), target velocity ($\dot{x}$, $\dot{y}$, $\dot{z}$), suspected target location ($suspected\_x$, $suspected\_y$, $suspected\_z$, $suspected\_r$, $suspected\_t$) and target id ($target\_id$). These comprise the parameters available for the `TrackTarget` interaction. Section 5 discusses the algorithm for deducing and publishing the target location from a simulated camera.

4. Controller

The UAV chosen for this project was the STARMAC, a quadrotor UAV being developed by a group at Stanford University. Both a description of its dynamics and a demonstration of its abilities can be found in [Hoffmann et al. 2007]. For easy visualization and to take advantage of Mathworks design tools (such as the SISO Design Tool), Simulink was preserved as the modeling language for designing the controller.

(a) An example sprite from the catalog, which does not represent our vehicle.



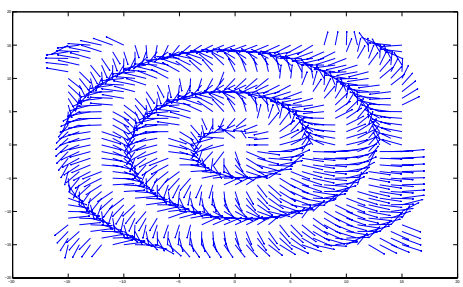
(b) 3D rendering of the STARMAC helicopter, used in Delta3D.

Figure 3. Rendered 3D models that can be used in Delta3D. Note that we are now able to use a sprite that represents our actual vehicle, based on a model developed in the Blender tool, and the actual dynamical model in Simulink.

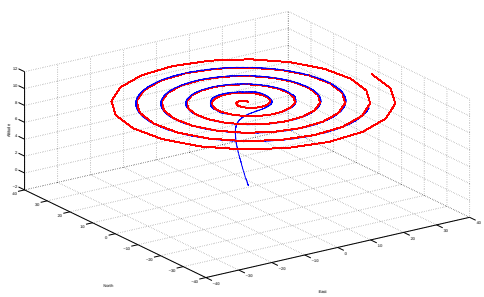
In order to respond to the varying command and control commands of the `Ground Station`, the Simulink controller can switch between various control laws. A top level view of the modified Simulink block diagram can be found in Figure 5. Both the waypoint and spiral search controllers can be seen on the left, with an input flag that specifies which one should be active.

The waypoint controller for the STARMAC was achieved using a 3rd order controller across the motor voltage command and setting up feedback loops around translational acceleration, velocity, and position.

The search algorithm for the UAV was chosen to be in the form of a spiral, originating and spiraling out from whatever location the UAV was at when the flag was switched over. The velocity vector field for the spiral/search guidance algorithm can be found in Figure 4(a). Ideally, the UAV should fly along the spiral. In order to achieve this, the algorithm takes the UAV position as an input, and returns the velocity it would like the UAV to achieve. Given a UAV with a position not on the spiral, the velocity issued is dependent on the distance from the nearest point. The farther the UAV is from the point, the more its velocity will be commanded directly towards that location on the spiral. As the craft nears the spiral, more and more of it's velocity vector is directed tangential to the spiral, until eventually the craft merges with the spiral and tracks it. A typical response can be found in Figure 4(b).



(a) Vector field showing spiral path following.



(b) Dynamic simulation of STARMAC following a spiral path in Simulink.

Figure 4. The controller design, done in Simulink, allows for component-scale simulation and analysis. In (a) the vector field shows the controller's response based on position (this represents the analysis and design phase.). In (b) a simulation of the controller from an initial condition below the spiral's origin is shown (this is the component-scale simulation).

The importance of these plots to this work is that they were the results of design done *inside* the design and simulation toolchain of Mathworks, including the ability to generate plots, utilize summing and feedback functions, and a fundamentally model-based approach. If we had decided to prototype the designs in MATLAB/Simulink and later port them to C/C++, errors could have been introduced either by changing to those languages (subtle errors of models of computation) or typographical errors (explicit errors not easy to find). With the integration of MATLAB/Simulink into the HLA infrastructure, we were able to use the same models developed in design and analysis as executable models.

5. Ground Feature Detection

For unmanned vehicles outfitted with a camera sensor, one potential application is unmanned and autonomous surveillance. The UAV could be given an instruction as, for example, "find the blue trucks in some area." Analyzing the output image from the camera is a crucial operation for the UAV to complete this task. It would have to determine first whether there was a blue truck in its field of view and report the rough location of that truck in some meaningful way, such as GPS coordinates, based on the image and its present state.

In order to closely emulate an actual implementation it is necessary to have a picture. However, simulation in Delta3D has a major drawback with respect to the sensor: the image cannot be directly analyzed. Therefore a software workaround is required to determine whether an object of interest is contained within the image. Taking advantage of the fact that knowledge of the simulation world is absolute, it is possible to bypass the image analysis phase. Instead of locating an object via the camera's image, an algorithm (given the known position of an object) can report where that object would appear in the image, if at all. From there the final task of meaningfully reporting that location is identical as if the pixel location came from an actual picture. Such an approach allows easy transition from canned simulation data to data obtained from analysis.

The process starts by feeding the algorithm the position of an object from Delta3D in x,y,z coordinates. Additional required data are the x,y,z coordinates of the UAV, its roll-pitch-yaw orientation, and the intrinsic parameters of the camera. The most important properties of the camera are its focal length, the size of the CCD, and the resolution. Given these, the algorithm produces the i,j pixel coordinates that represent where in the picture the specified object lies. These coordinates are then adjusted by some noise factor to simulate receiving actual data. Finally the adjusted pixel coordinates are reverse transformed and reported as an approximate location for the object. In practice only the latter portion is necessary because the image will actually be available for analysis.

To accomplish this, Delta3D and MATLAB/Simulink must communicate with one another. Delta3D provides MATLAB the x,y,z coordinates of the object and UAV while Simulink provides the state and orientation data directly to the camera (i.e., not published through the RTI). The camera parameters are fixed and so are simple constants. All the calculations are carried out in MATLAB. An overview of the process can be seen in Figure 6.

6. Integration

Section 2 describes how the GME Environment was used to develop an application model with the help of the HLA paradigm. The `UAV` and the `Ground Station` federates and the associated interactions were discussed, including how the UAV is controlled through Simulink. Target detection was discussed in Section 5, and requires state information of the `UAV` as well as information regarding the target's location.

Now with each of these pieces developed, simulated, and tested individually, we should integrate them into a demonstration. To do this, we follow the overall structure as described in Figure 7. Using Portico as the RTI infrastructure, we use a Java implementation of a `Ground Station` which provides human command input into the simulation. These commands include direction for the `UAV` to search for a target, fly to a waypoint, track a target, etc. The other Java implementation (a work in progress) publishes information about the location of a target, in order for the MATLAB component discussed in Section 5 to publish information about a target being acquired.

These Java-based components express basic control-flow, and also have their own GUI, utilizing Java's user-interface libraries.

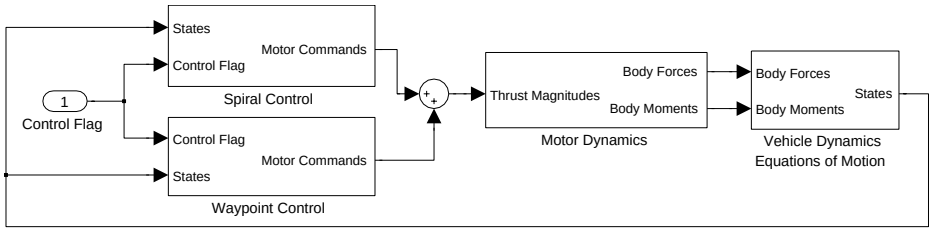


Figure 5. Top-level block diagram of Simulink STARMAC model. Note that the switching controller on the left side of the figure represents the hybrid behavior of the controller, whether the vehicle is flying the spiral search pattern, or the waypoint following pattern.

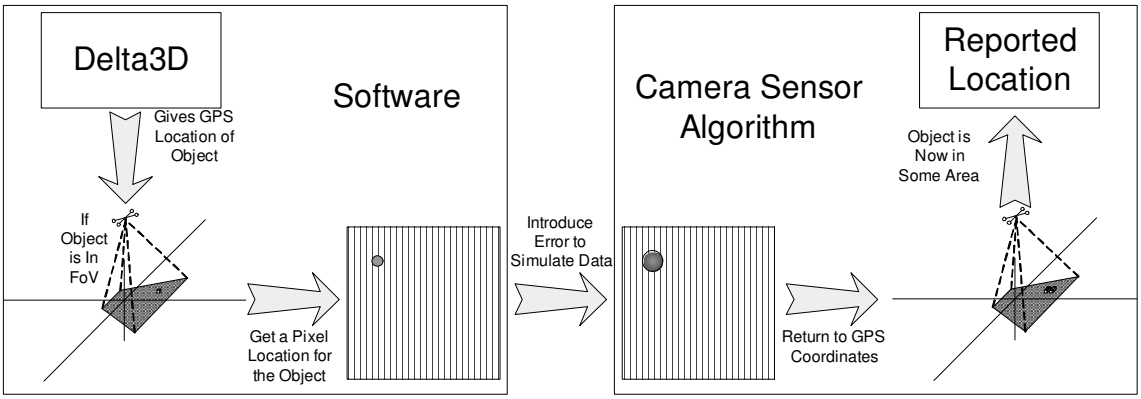


Figure 6. Overview of GPS target identification scheme, given that no image processing is available. If the object is in the field of view (FoV), it is passed along to the targeting component to report location.

For the physical-system simulation, the Simulink models discussed in Section 4 are called, which publish updated position information. This positional information, as well as location of the target, are read by the Delta3D component, and visualized for the benefit of the `Ground Station` human operator. The final result is an

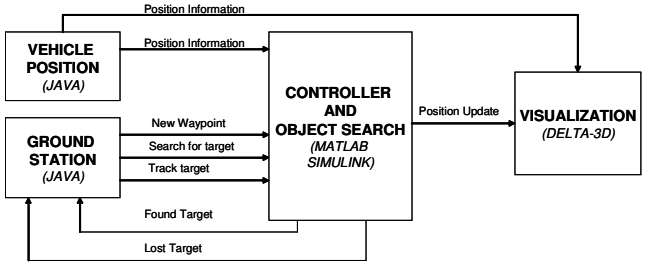


Figure 7. Integrated Schema of the Federation Execution

integrated demo that can be run from a Windows `.bat` file, reducing the possibility of human error in starting up components in the wrong order, or forgetting to pass in parameters. Such an automation for running the demonstration also reduces the effort required to run tests to confirm that certain tools (e.g., MATLAB) are properly integrated into the demonstrator's machine.

7. Results and Analysis

We successfully integrated several demonstrations that showed our various technical contributions. Depending on the number of interactions that we utilized in each demonstration model, about 5000 lines of software were generated for the entire set of federates available. This included the standard "getter" and "setter" methods for various objects, but more importantly the "publish" and "subscribe" methods were provided, reducing the complexity of programming for domain experts. For the Simulink interaction, some hand-editing of the model is required to integrate, i.e., replacing the state reading and writing blocks with HLA reading and writing blocks. This is important not just for information exchange, but also to prevent Simulink from advancing more rapidly than other portions of the simulation, and thus not synchronizing data with other components.

Based on the amount of generated code discussed in Section 7, it would require a *significant* amount of human effort to code the various integration points for each tool. The HLA modeling language provided an integration point for *each* software tool we needed, as well as many others that we did not need. This not only provides a late-stage integration freedom, but also gives a design freedom, where alternative tools can be explored in parallel tested upon integration for selection of the optimal behavior. In addition to the raw effort of programming the interaction points, there is significant effort required to understand *how* the tools could interact with the middleware. Thankfully, this task has already been done by the HLA modeling language designers.

There are, however, several areas in which the tool can be improved. As of now, the integration of components running on different machines is performed through shared drives. This could be improved to use TCP/IP across a network. To mitigate this shortcoming, such integration is currently performed through code generation, so a better integration solution will also be transparent to the users.

Another area for improvement is the integration with MATLAB/Simulink, which currently requires some user editing the MATLAB/Simulink model to include the generated interfaces to HLA. We leave this solution up to the language designers, though

one possible approach is to generate a library of blocks that can be used, and then updates to models in these blocks will automatically update any simulation models.

8. Conclusions and Future Work

In under three months, the authors were able to integrate a new demonstration of C2 behaviors, including new controllers for the quad-rotor vehicle, new commands sent to the vehicles, new models of the demonstration, and summary simulations that verify behavior on a new installation of the infrastructure. These summary simulations are important for a distributed team, as they confirm to other team members that various functional components are behaving correctly, and also confirm to those teams that they can run the simulation tools required.

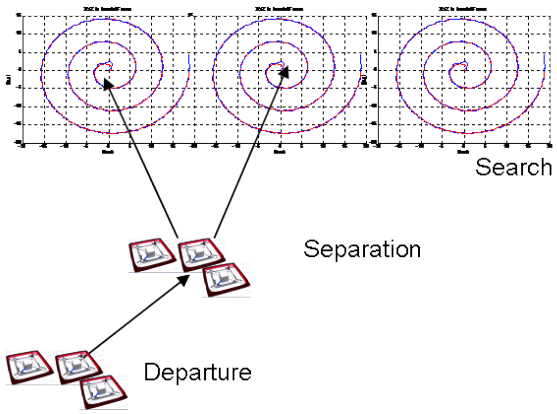


Figure 8. Co-operative Search Operations

Our future work includes development of high-level control algorithms for managing a group of vehicles that co-operatively search for target(s) at a specified location(s). This would be an implementation of mixed-initiative control. The key issues would involve ensuring a stable formation and generating optimal search algorithms. The UAVs would depart as a group in response to a command, and would separate mid-way to perform individual search operations spanning the entire search area, as in Figure 8. Dividing the search space optimally, avoiding collisions and reporting back appropriate information would require the inclusion of intelligent real-time algorithms in the controller. Mesh stability is a good model to obtain a stable formation, as it attenuates disturbances acting on one vehicle as they propagate to other vehicles. Thus the UAVs travel in a mesh. This calls for decentralized control laws and intelligent search strategies.

DSMs present a significant advantage in the high-level specification of system interaction, especially when the generation of the software that produces their interaction (i.e., the "glue-code" that holds an interaction together) is computational, and not a case-by-case design. We believe that future uses of domain-specific modeling environments in this domain will further enable experts in control, visualization, computer vision, etc., to put experiments of system-level simulations together more easily than a brute-force integration strategy.

Acknowledgments

This work would not have been possible without the efforts of Gyorgy Balogh, Himanshu Neema, Harmon Nine, Gabor Karsai, and Janos Sztipanovits of the Institute for Software Integrated Systems. Special thanks are due to Gabe Hoffman, Claire Tomlin, and Hal Tharp for their generous advice in the development of the quad-rotor controllers. This work is supported by the Air Force Office of Scientific Research, under award #FA9550-06-1-0267, titled "Human Centric Design Environments for Command and Control Systems: The C2 Wind Tunnel".

References

Gyorgy Balogh, Himanshu Neema, Graham Hemingway, Jeff Green, Brian W. Williams, Janos Sztipanovits, and Gabor Karsai. Rapid Synthesis of HLA-Based Heterogeneous Simulation: A Model-Based Integration Approach. *Unpublished manuscript*, 2008.

Gabriel M. Hoffmann, Haomiao Huang, Steven L. Wasl, and Claire J. Tomlin. Quadrotor helicopter flight dynamics and control: Theory and experiment. In *Proc. AIAA Guidance, Navigation, and Control*, 2007.

IEEE-HLA-1516. IEEE standard for modeling and simulation high level architecture (HLA)-framework and rules. *IEEE Std 1516-2000*, pages i–22, Sep 2000.

K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3–4):213–254, 2007.

A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *IEEE Computer*, 34(11):44–51, 2001.

A. Ledeczi, G. Balogh, Z. Molnar, P. Volgyesi, and M. Maroti. Model integrated computing in the large. *Aerospace Conference, 2005 IEEE*, pages 1–8, March 2005.

Sivakumar Palaniappan, Anil Sawhney, and Hessam S. Sarjoughian. Application of the DEVS framework in construction simulation. In *WSC '06: Proceedings of the 38th conference on Winter simulation*, pages 2077–2086. Winter Simulation Conference, 2006. ISBN 1-4244-0501-7.

András Varga. The OMNeT++ Discrete Event Simulation System. In *Proceedings of the European Simulation Multiconference*, June 2001.

A Family of Languages for Architecture Description

Markus Voelter

Independent Consultant, Grabenstrasse 4, 73033 Goeppingen, Germany
voelter@acm.org, http://www.voelter.de

Abstract

In this paper I describe how product line engineering and variant management can be applied to domain-specific languages. I introduce concepts and a tool prototype for describing a family of DSLs used for architecture description. I want to make two points in this paper: First, I want to introduce the idea of product line engineering for domain-specific languages, and second, I want to illustrate why and how this approach is especially useful for DSLs that describe software architectures. The paper is based on practical experience and not on academic research.

Categories and Subject Descriptors D.2.2 Design Tools and Techniques, D.2.11 Software Architectures, D.2.13 Reusable Software, D.3.3 Language Constructs and Features

General Terms Documentation, Design.

Keywords software architecture; domain-specific languages; variability management; product line engineering

1. Overview

Architecture DSLs

Architecture is typically either a very non-tangible, conceptual aspect of a software system that can primarily be found in Word documents, or it is entirely driven by technology ("we use an XML architecture"). Both are bad: the former makes it hard to work with, and the latter hides architectural concepts behind technology hype.

What can be done? As you develop the architecture, evolve a language that allows you to describe systems based on this architecture. Based on my experience in a number of real-world projects, this makes the architecture tangible and provides an unambiguous description of the architectural building blocks as well as the concrete system while still staying away from technology decisions (which then can be made consciously in a separate step).

In other words, I am advocating the use of DSLs to describe the architecture of a specific system or product line.

The beauty of textual languages

I like to use textual languages for this endeavor, for the following reasons:

- First of all, languages as well as nice editors are much easier to build compared to custom graphical editors (e.g. those built with Eclipse GMF)
- Textual artifacts integrate much better with existing developer tooling compared to graphical models based on some kind of repository. You can use the well-known diff/merge tools, and it is much easier to version/tag/branch models and code together.
- Model evolution (i.e. the adaptation of the models in cases where the DSL evolves over time, something you'll always have in practice) is much simpler. While you can use the standard approach – a model-to-model transformation from the old version to the new version – you can always use search/replace or *grep* as a fallback, a technology familiar to basically everybody.
- Lastly, textual DSLs are often more appreciated by developers, since "real developers don't draw pictures".

Graphical notations are useful, of course. Whenever you want to show the relationship between entities a graphical notation is potentially better suited. Also, whenever you want to communicate to non-technical people, graphical languages are typically preferrable because they are perceived to be "easier to understand".

However, there is a different between graphical *notation* and graphical *editing*! Using tools like Graphviz [13] or Prefuse [14], you can easily render a textual model in a graphical way – without being able to *edit* in the graphical environment. Since the model contains the relevant data in a clear and unpolluted form, you can easily transform the model data into a form that tools like GraphViz or Prefuse can process.

The following is an example of a graphviz-generated diagram. It shows namespaces, components, interface, datatypes as well as the dependencies between those.

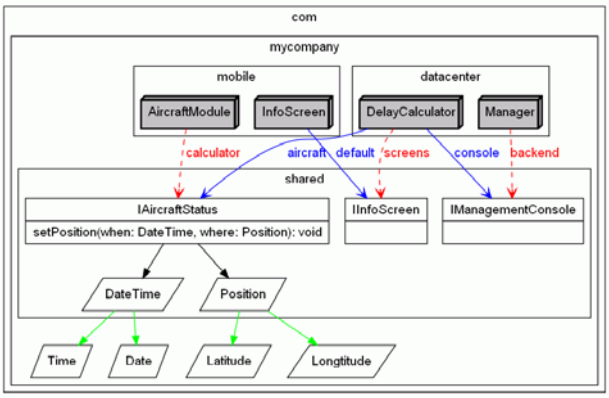


Figure 1. Visualization via Graphviz

The challenge of reuse

I argue above that it is essential that the architecture DSL is developed as you understand your architecture, i.e. that it is specific to the system at hand. Just using an existing, generic architecture description language (such as UML or one of the many ADLs) does not reap the same benefits because you have to shoehorn your domain's architecture into the existing modeling language.

However, that does not mean that there aren't a number of architectural features that are found in many different systems or projects. It is not sensible to put all of them into a "generic architecture DSL", but it is sensible to make it trivially simple to add the respective feature to the DSL once you've identified it as being relevant to your architecture.

Enter product lines: I advocate building a product line of architecture DSL where we use feature modeling to capture the variations. So, instead of doing feature model-based variability management on the concrete system level, we do it on language (grammar) level.

Note that the approach to variant management for languages is of course not technically limited to architecture DSLs. However, specifically in case of software architecture there's a high degree of similarity between different systems, hence the potential and need for reuse is especially high.

Code Generation

Once we have defined a modeling language that accurately reflects the architecture of a software system, and once we have described actual systems with this language, we have to decide what to do with the models. Generally it is my firm belief that if you *don't do something* with your formal models, they are pretty useless (actually, in this case, the process of describing the architecture and the systems has a value in itself, since it helps you understand your own systems much better).

Hence, we will generate two kinds of code models: API code is used by developers to implement manually written business against. Glue code adapts the API code and the manually written code to some kind of implementation/middleware platform.

A central point of this paper is that we describe variants of the architecture DSL. Consequently we also need to vary the code generator, typically based on the same configuration features that are used to define variants of the language.

openArchitectureWare includes a number of features useful to this end. For example, you use aspects for code generation templates, model transformations and workflow specifications to define variants of code generators. The deployment of the aspects can be made to depend on configuration features, too, making the language as well as it's processors depend on the same configuration model. I don't describe this any further in this paper, but you can read more about it in [15].

The structure of this paper

The rest of the paper is structured as follows. Section 2 contains a discussion of how to build an architectural meta model for component-based architectures and also shows a set of typical variations that I came across over the years. It is those variations we'll capture in the feature model. Section 3 looks at how language tooling can be implemented to be able to express the variability discussed in section 2. Section 4 then looks at how users use this variability to configure and customize their own language. Section 5 looks at the current state of the prototype, and section 6 contains an evaluation as well as directions for future work.

2. A Product Line for Component Meta Models

This section introduces a set of typical features of a component DSL. I'll start with defining a set of basic viewpoints and their meta models. The rest of the section then looks at variations of those viewpoints/meta models. These have been extracted from years of work building component-oriented architectures, most of them using formal modeling as a basis.

Viewpoints

A viewpoint describes a specific aspect or concern of a system. It has a limited number of connections to other viewpoints, making each of the viewpoints reusable and well modularized. We use three viewpoints as the foundation for component architectures: type viewpoint, composition viewpoint and system viewpoint. Again, those viewpoints are based on experience gained in many projects over the years. They can also be found in various industry standards (such as EJB or SCA), even if they are not necessarily explicitly distinguished and given names.

Type Viewpoint

The type viewpoint describes component types, interfaces, and data structures. A component provides a number of interfaces and references a number of required interfaces. An interface owns a number of operations, each with a return type, parameters, and exceptions. Alternatively, for message oriented systems, an interface can also be a collection of messages, where a message is named and has a number of parameters. In this case, the interface also defines the direction (in/out) of messages, or even message interaction patterns (*oneway, request-reply, publish-subscribe*)

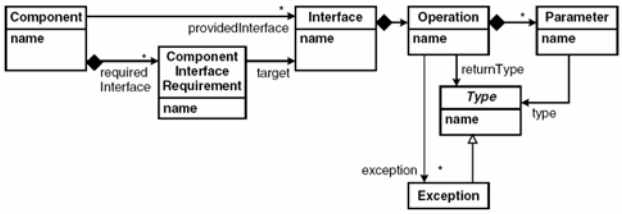


Figure 2. Type viewpoint, components

To describe the data structures with which the components work, we start out with the abstract type Type. We use primitive types as well as complex types. A complex type has a number of named and typed attributes. There are two kinds of complex types. Data transfer objects are simple *structs* that are used to exchange data among components. Entities have a unique ID and can be made persistent (this is not visible from the meta model). Entities can reference each other and thus build more complex data graphs. Each reference has to specify whether it is navigable in only one or in both directions. A reference also specifies the cardinalities of the entities at the respective ends.

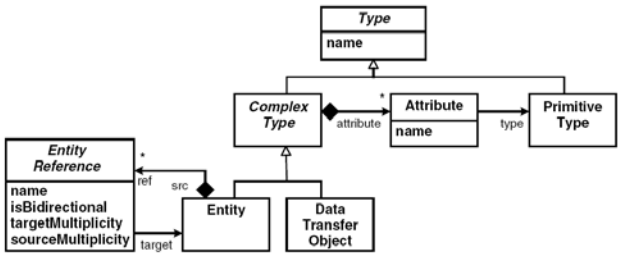


Figure 3. Type viewpoint, data

This data meta model is not much different from the entity relationship model or the SDO standard. In many scenarios, the data meta model can probably be simplified quite a bit, basically reducing it to the equivalent of Java beans or C structs.

The Composition Viewpoint

This viewpoint, illustrated in the following diagram, describes component instances and how they are connected. A *configuration* consists of a number of component *instances*, each referencing its type. An instance has a number of *wires*: a wire is an instance of a *component interface requirement* and hence a connector between component instances. Note the constraints defined in the meta model:

- For each component interface requirement defined in the instance's type, we need to supply a wire.
- The type of the component instance at the target end of a wire needs to provide the interface which the wire's component interface requirement references.

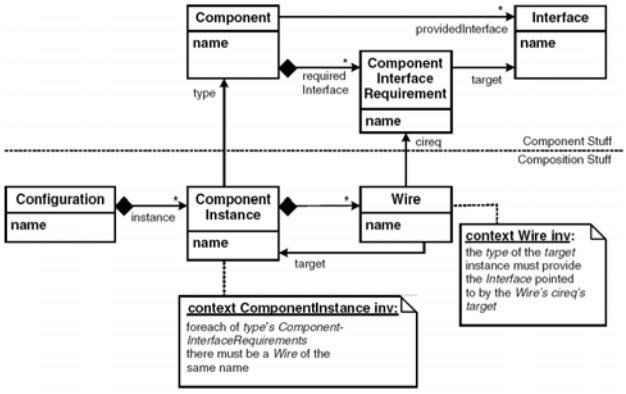


Figure 4. The composition viewpoint

Using the type and composition viewpoints, it is possible to define component types as well as their collaborations. Logical models of applications can be defined. From the composition viewpoint, you can generate or configure a container that instantiates the component instances. Unit tests that verify the application logic can be run in an infrastructure-free environment.

The System Viewpoint

This third viewpoint describes the system infrastructure onto which the logical system defined with the two previous viewpoints is deployed. A *system* consists of a number of *nodes*, each one hosting *containers*. A container hosts a number of component *instances*. Note that a container also defines its kind – this could be things like CCM, J2EE, Eclipse, Spring or a proprietary runtime infrastructure.

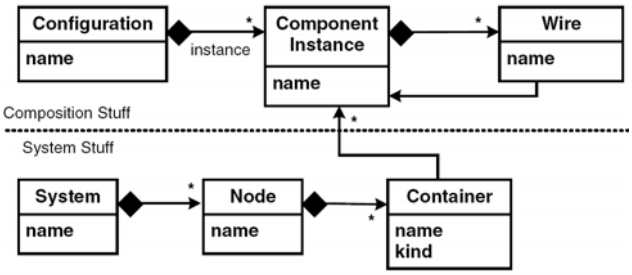


Figure 5. System viewpoint

Based on this information, you can generate the necessary glue code to run the components in that kind of container. The node information, together with the wires (connections) defined in the composition model, allows you to generate all kinds of things, from remote communication infrastructure code and configuration to build and packaging scripts.

Viewpoint Dependencies

You may have observed that the dependencies among the models (and meta models) are well-structured. Since you want to be able to define several compositions using the same components and interfaces, and since you want to be able to run the same compositions on several infrastructures, dependencies are only allowed in the directions shown in the next diagram.

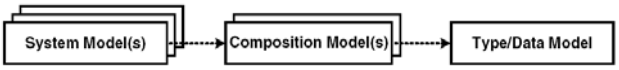


Figure 6. Viewpoint dependencies

Variations

The meta models we describe above cannot be used in exactly this way in every project. Also, in many cases the notion of what constitutes a component needs to be extended. As explained earlier, it is essential that the DSL for describing an architecture evolves and grows with the architecture itself. However, there are common variations. In this section we illustrate some of these.

Separate Interfaces

You might not need separate interfaces. Operations (or messages, respectively) could be owned directly by the components. As a consequence, of course, you cannot reuse the interface "contracts" separately, independently of the supplier or consumer components.

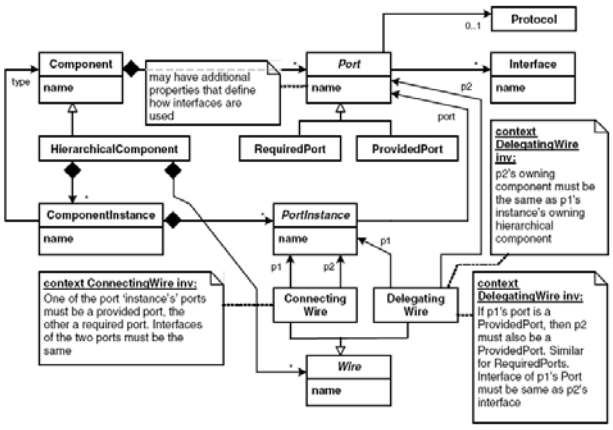


Figure 7. Hierarchical Components

Hierarchical Components

Hierarchical components are a very powerful concept: a component is internally structured as a composition of other component instances. Ports define how components may be connected: a port has an optional protocol definition that allows for port compatibility checks that go beyond simple interface equality. While this approach is powerful, it is also non-trivial, since it blurs the formerly clear distinction between type and composition viewpoints.

Configuration Parameters

A component might have a number of configuration parameters – comparable to command line arguments in console programs – that help configure the behavior of components. The parameters and their types are defined in the type model, and values for the parameters can be specified later, for example in the composition or the system models, or through configuration files.

Component Kinds and Layering

Often you'll need different kinds of components, such as domain components, data access (DAO) components, process components, or business rule components. Depending on this component classification you can define constraints that check whether certain component dependencies are valid or not. You will typically also use different ways of implementing component functionality, depending on the component types. In effect, this gives you a way of layering application functionality.

Another way of managing dependencies is to mark each component directly with a layer tag, such as *domain*, *service*, *gui*, or *facade*, and define constraints on how components in these layers may depend on each other.

State, Threads and Lifecycle

You might want to specify something about whether the components are stateless or stateful, whether they are thread-safe or not, and what their lifecycle should look like (for example, whether they are passive or active, whether they want to be notified of lifecycle events such as activation/passivation, and so on).

Communication Paradigm

Even if a decision has been made for RPC-style communication, it is not always enough to use simple synchronous communication. Instead, one of the various asynchronous communication patterns, such as those described in the *Remoting Patterns* book [16], might be applicable. Because using these patterns affects the APIs of the components, the pattern to be used has to be marked up in the type model.

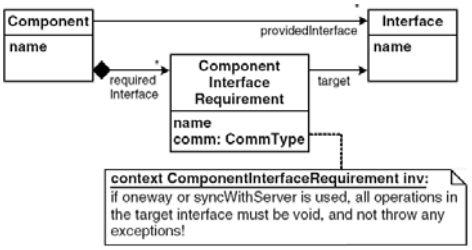
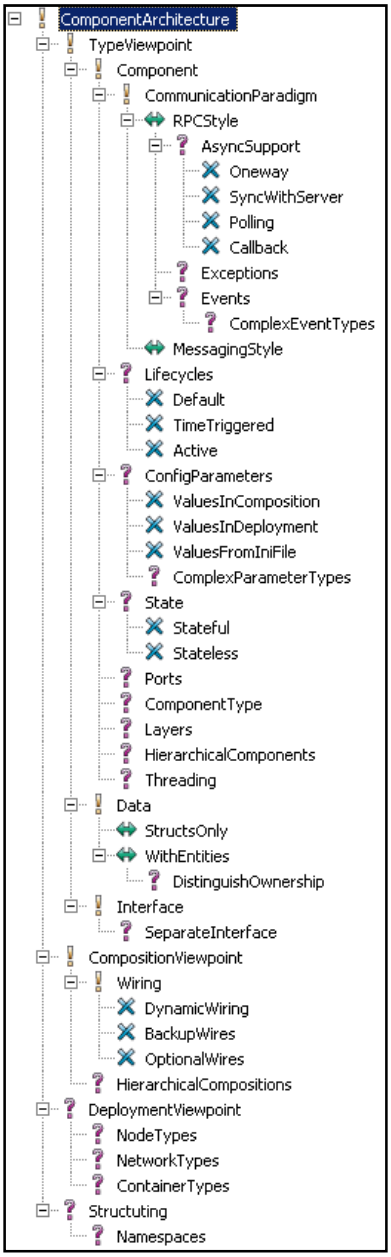


Figure 8. Communication Alternatives

Of course when messaging is used the communication is asynchronous anyway. However, even in that case it makes sense to capture a set of predefined communication paradigms such as *oneway*, *request/reply* or *publish/subscribe*.

Events

In addition to the communication through interfaces, you might need (asynchronous) events using a static or dynamic publisher/subscriber infrastructure. It is often useful that the "direction of flow" of these events is the opposite of the *uses* dependencies discussed above, i.e. they propagate from the *used* entity to the *using* entity.



Static vs. Dynamic Connection

The composition model connects component instances statically. This is not always feasible. If dynamic wiring is necessary, the best way is to embed the information that determines which instance to connect to at runtime into the static wiring model. So, instead of specifying in the model that instance A must be wired to instance B, the model only specifies that A needs to connect to an instance with a specific set of properties: it needs to provide a certain interface, and for example offer a certain reliability. At runtime, the wire is "dereferenced" to a suitable instance using a repository/naming/lookup/trader service.

Higher Level Structures

Finally, it is often necessary to provide additional means of structuring complex systems. The terms *business component* or *subsystem* are often used. Such a higher-level structure consists of a number of components. Optionally, constraints define which kinds of components may be contained in a specific kind of higher-level structure. For example, you might want to define that a business component always consists of exactly one façade component and any number of domain components.

3. PLE for Languages – Tool Implementation

This section explains how to conceptually implement the architecture DSL product line approach for a textual language. We use Eclipse [1], EMF [2], openArchitectureWare [3] and pure::variants [4] as tooling. Specifically, openArchitectureWare's Xtext is used for the textual editor. The way it works is that you specify the grammar for your language, and the meta model, parser and editor are automatically derived from that grammar. In addition, you also have to specify constraints.

Feature Modeling

Feature modeling is used to describe the variability of the architecture DSL. The tall diagram on this page shows a pure::variants feature model with some of the variability mentioned in the previous section.

Based on this feature model, the architecture DSL can be adapted to the needs of a specific architecture as it arises.

Of course, there are facilities that allow for custom configuration, i.e. to put features into the architecture DSL that are not available as a simple configuration option from the feature model.

Variability Mechanisms for Textual Languages

It is not enough to describe conceptual variability in feature models. It is similarly important to actually implement the variability in the artifacts for which we define variations.

In the case described here we want to vary the definition of the architecture DSL (grammar, constraints) as well as the respective editor (code completion, outline, etc.).

In a scenario where the respective artifacts are built with openArchitectureWare's Xtext, this requires variation of the following artifacts: Xtext grammar definitions, check files and extension files. As of now, none of those kinds of files can contain explicit feature dependencies – those artifacts do not "know" they are being varied.

Consequently, we have to use low level "text modification" based on the features. This is similar to Gears' [5] way of implementing variability and is basically a generalized C *#ifdef*. Feature-dependency is expressed with special comments:

```
//# SomeFeature
context Component ERROR "error message":
  here.is.the.actual.contraint.condition;
context Configuration ERROR "another message":
   constraint > 0;
//#~ SomeFeature
```

A preprocessor takes the files marked up with those comments and removes everything for which the corresponding feature is not selected. The marked up file itself contains all possible alternatives (hence this is a form of negative variability).

In the current implementation of our tooling, there is some integration between the text editors and pure::variants:

- Feature names mentioned in artifact files are statically cross-checked with the feature model. If you mention a feature name that is not in the feature model, you'll get an error in Eclipse's Message view.
- Also, you can select any feature in the feature model and see in pure::variants Relations view in which of your artifact files it is referenced.

Customization

Again let me emphasize that is it important to be able to directly represent the architectural concepts of a *specific* system in the architecture DSL. It is therefore not enough to "just" configure a DSL from a set of predefined configuration options, even if these are typical, and hence likely to be a good starting point for your specific system. It is still necessary that the DSL developer can customize the DSL with arbitrary additional grammar.

This is easily possible. The grammar derived from the feature model shown above will contain hooks in various places where customization can happen. It is again based on "text mangling". We show an example in the next section.

4. Using the tools

This section explains how to use the tooling from the perspective of a DSL developer or architect.

Configuring your language

Open the configuration model and select the options you want for your language. In the example here, we want to be able to express stateful components and use exceptions for the operations of our interfaces. We select the *Stateful* and *Exceptions* features as shown in the illustration below.

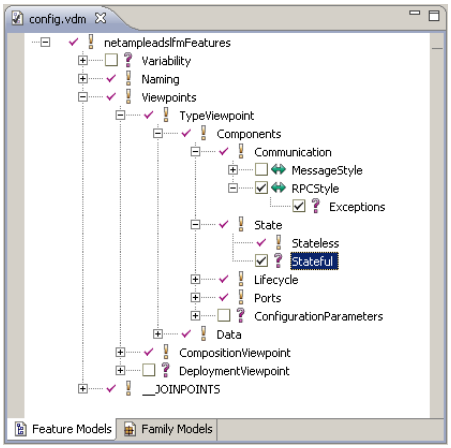


Figure 9. Selecting Stateful Components with Exceptions

Then we regenerate the language (by running the *configureMyLanguage.oaw* workflow) and rebuild the tooling (running *generateDSLAndTooling.oaw*). We are now able to use the following notation in the language-aware editor:

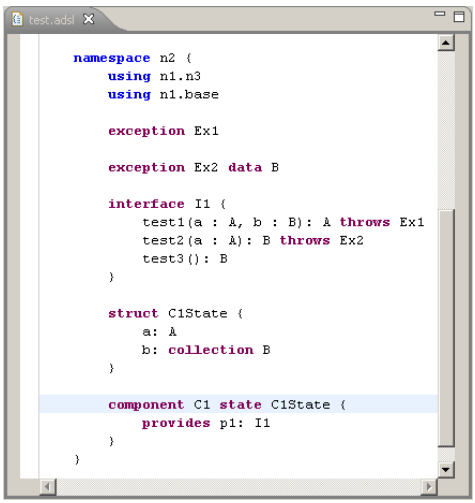


Figure 10. Resulting Language and Editor

Customizing the Language

We call *configuration* the activity of adapting your language by selecting features from the configuration model. This is in contrast to *customization*, by which we mean the extension of the ADSL with your own, project specific features.

There are two fundamentally different ways to go about this: you can define an external viewpoint, i.e. a completely separate language that "annotates" an ADSL model. Technically, this is not an adaptation of ADSL, but can serve the same purpose. Al-

ternatively you can also extend the actual ADSL by adding custom code to a set of predefined hooks.

External Viewpoint

An external viewpoint is especially useful if you want to describe something that relates to the abstractions defined in the ADSL model, but is sufficiently different for it to be expressed with a different notation or by a different role in the development process.

Since Xtext models can be treated as an EMF resource, you can, with the facilities provided by EMF, reference an Xtext model element from any other EMF model and hence "annotate" it. The following example shows how to define another textual DSL to annotate the ADSL model.

Assume you want to define some kind of database mapping for your data structures. To do that, you define a separate DSL using the following piece of code as the grammar:

```
importMetamodel "platform:/resource/
   net.ample.adsl.language/src-gen/net/ample/adsl/
   language/adsl.ecore" as adsl;

DBMapping:
        (imports+=Import)*
        (structMappings+=StructMapping)*;

Import:
        "adsl" file=URI;

StructMapping:
        "map" struct=[adsl::ComplexType|ID] "{" "}";
```

In this grammar we import the generated meta model of the ADSL language and reference elements from it (in the *StructMapping* rule). Once you generate the editor and run it the editor provides code completion and constraint checking into the ADSL file, thereby providing tight integration between the two languages.

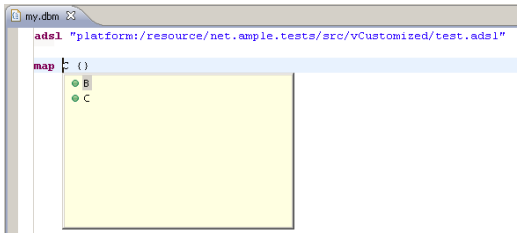


Figure 11. Code-completing into the ADSL file

By loading the persistence model as well as the ADSL model, you can now generate code from both models together – effectively customizing the ADSL model without touching the language itself.

In-Language Extension

The other approach to customization is the custom extension of the ADSL language itself. You can extend the language in all respects, you can even change parts of the configured language. Language customization happens in three steps:

- In the base language (the one you get via configuration) a hook must be defined in all the location where customization is intended. Borrowing from AO, we call these hooks joinpoints.
- You can then define advices (again, borrowing from the AO terminology) that contribute additional code *before*, *after* and *instead of* predefined joinpoints.
- Step three is the execution o the weaver, which actually contributes the advices to the joinpoints.

Let us look at an example. Imagine you want to be able to embed a statemachine in a component. The grammar for a statemachine is probably relatively straight forward. Here is one way of integrating state machines into component grammar:

- We need to add a reference to a statemachine inside a component
- And we'd need to embed the actual statemachine as a top level content in the namespace.

Defining the Joinpoints: We start by defining those two joinpoints in the original overall grammar. In the following piece of grammar, the joinpoints are highlighted in bold (the lines beginning with //>).

```
Namespace:
      "namespace" name=ID
   //# NamespaceFeatureDependencies
                    (featureClause=FeatureClause)?
        "{"
              (usings+=Using)*
              ( subNamespaces+=Namespace
                | components+=Component
                | datatypes+=DataType
                | interfaces+=Interface
                | compositions+=Composition
                //# DeploymentViewpoint
                     | systems+=System
                //# Exceptions
                     | exceptions+=Exception
                //> AdditionalNamespaceContents
                //~> AdditionalNamespaceContents
               )*
      "}";

Component:
      "component" name=ID
      //# Active (isActive?="active")?
      //# Periodic (isPeriodic?="periodic"
                 "(" period=INT ")")?
      //# Stateful ("state" state=[ComplexType|ID])?
      "{"
              (ports+=Port)*
          //> ComponentContentsAfterPorts
          //~> ComponentContentsAfterPorts
      //# ConfigurationParameters
              (configuration=
                   ComponentConfiguration)?
      "}";
```

These joinpoint markers will end up in the generated, configured grammar; note how we use the comment to make the joinpoints invisible to the grammar generator. Note also, that the tooling provides checks against the feature model, so if you refer to a joinpoint that is not defined, you'll get an error message:

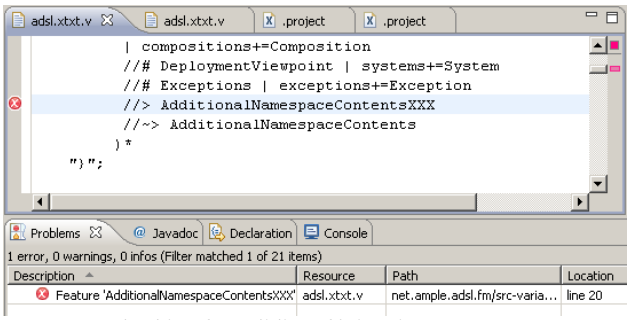


Figure 12. Checking for validity of joinpoints

Note that these joinpoints are not really *configuration* features – however, we still use the feature model to uniquely define the *names* of those joinpoints.

Defining the advice. For the statemachine example, we have to define a number of advices. First of all, we need to define the statemachine itself in the grammar. Note in bold the actual advice syntax. In this case, we put something after (i.e. at the end of) the *TopLevelContents*.

```
//+ after:TopLevelContents
Statemachine:
      "statemachine" name=ID "{"
              (states+=State |
               events+=Event)*
      "}";

State:
      "state" name=ID "{"
              (transitions+=Transition)*
      "}";

Transition:
      event=[Event|ID] "->" target=[State|ID];

Event:
      "event" name=ID ":" operation=[Operation|ID];
//~+
```

To add a statemachine to a component, we need to advice the *ComponentContentsAfterPorts* joinpoint:

```
//+ after:ComponentContentsAfterPorts
      (statemachine=Statemachine)?
//~+
```

Both of these advices are located in a separate file */demo.config/src/adsl.xtxt.v*. The file has the same name as the file into which it is woven into, plus the *.v* extension which is used for all variant files.

We also define a constraint that checks the uniqueness of state names in a statemachine. These need to be contributed to the *net::ample::adsl::language::Check.chk* file, which defines a joinpoint *TopLevelContents* for this purpose. Here's the advice, which, as you might expect, is in the */demo.config/src/net/ample/adsl/language/Checks.chk.v* file:

```
//+ before:TopLevelContents
context State ERROR "State name not unique" :
    ((Statemachine)eContainer).states.
          select(s|s.name == name).size == 1;
//~+
```

After running the "text file weaver", the result is an ADSL version that supports embedded statemachines:

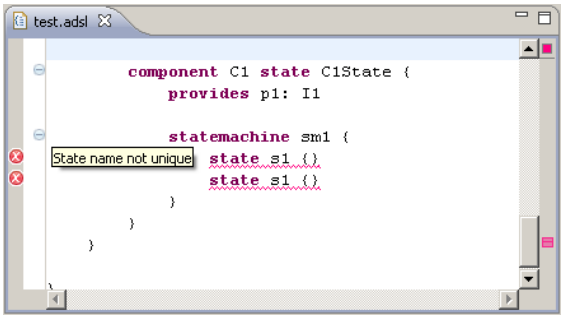


Figure 13. Resulting editor, with state machines

5. The state of the prototype

The prototype has been developed as part of the AMPLE project and is in the process of being made open source. We're looking for interested parties to help develop it further. If you're interested, please contact the author.

The tooling is generally done, but the set of configuration features is limited (ca. 25 options as of now).

There is also a simple Java API generator that generates a mapping of the selected language features to Java; however, there is no generator yet for any specific target platforms.

We also have integrated visualization facilities using Graphviz (for printing) and Prefuse (for interactive visualization).

6. Evaluation, Related Work and Future Work

Since implementing the toolkit, I have used the toolkit for two other customers. We have selected the language features necessary for their architecture, generated the tooling, and used it for real project work. It is fair to say the approach works in practice.

Isn't a generic language good enough?

Describing architecture with formal languages is not a new idea. Various communities recommend using Architecture Description Languages (ADLs) or the Unified Modeling Language (UML) for describing architecture. Some even (try to) generate code from the architecture models. However, all of those approaches advocate using existing generic languages for documenting the architecture (although some of them, including the UML, can be customized).

I don't see much benefit in shoehorning your architecture description into the (typically very limited) set of constructs provided by predefined/standardized languages. The idea is to actually build your own language to capture your system's conceptual architecture. Adapting your architecture to the few concepts provided by the ADL or UML is not very helpful.

So this raises the general question about standards. Are they important? Where? And when? In order to use any architecture modeling language successfully, people first and foremost have to understand the architectural concepts they are dealing with. Even if the UML standard is used to do this people will still have to understand the concepts and map them to the language – in case of using UML that would be an architecture-specific profile. Of course, then, the question is whether such a profiled UML is still

standard. Also, I am not proposing to ignore standards generally. The tools are built on MOF/EMOF, which is an OMG standard, just like the UML, just on a different meta level.

A specific note on UML and profiles: yes, you could use the approach explained above with UML, building a profile as opposed to a textual language. I have done this in several projects and while it does work, my conclusion is that it doesn't work very well in most environments. Here are some of the reasons:

- Instead of thinking about your architectural concepts, working with UML requires you to think more about how you can use UML's existing constructs to more or less sensibly express your intentions. That's the wrong focus!
- Also, UML tools typically don't integrate very well with your existing development infrastructure (editors, CVS/SVN, diff/merge). That's not much of a problem if you use UML during some kind of analysis or design phase, but once you use your models as source code (they accurately reflect the architecture of your system, and you generate real code from them) this becomes a big issue.
- In today's tools, a UML profile cannot remove things the UML provides out of the box. Consequently, the meta model of the model you create is a superset of the (already non-trivial) UML meta model, making it even more complex. Since you want to process your models with generators or transformers, this meta model complexity is an issue to reckon with.
- Finally, UML tools are often quite heavyweight and complex, and are often perceived as "bloatware" or "drawing tools" by "real" developers. Using a nice textual language can be a much lower acceptance hurdle.

Related Work

Architecture Modeling. Using formal languages to describe software architectures is of course nothing new. UML is often used for this purpose, as are the many ADLs that are available on the market [6,7,8]. The approach of defining a domain-specific ADL can also be found elsewhere, an example is AUTOSAR [9] in the automotive world.

However, the approach advocated in this paper is based on defining an architecture DSL that is much more specific to the platform or system being built. The process of defining the language is integral to defining the architecture – architecture definition and language creation cross-polinate each other.

Language Customization. Many general purpose modeling languages provide some kind of customization. Many ADLs allow you to define new "component types" – basically a type label that can be associated with a component. This is a very simplistic approach that does not allow the definition of new architectural abstractions that come with their own structure, constraints and syntax. The approach described in this paper supports arbitrary configuration and customization of languages.

The best known example for language customization is of course the UML with its profile mechanism. I have already discussed this in the *Isn't a generic language good enough?* section above. The approach advocated in this paper tailors a language by actually *removing* features you don't need in a given scenario. Hence the editor, the meta model and all other subsequent model processing is simplified along with the language.

Language Modularization. Being able to define language modules and then integrate those modules into "composite languages" is of course an active area of research. This is a non-trivial problem, because you'll have to somehow combine the parsers. In some cases you'll have to regenerate a new parser based on the combined grammars (that will be the approach available in oAW 5, see below). In other environments (such as SDF, [10]) languages can be combined without regeneration of the composite parser.

Other language engineering environments support the modularization of languages without the need for a parser. Examples include MetaEdit+ [11] (which supports mainly graphical DSLs, where the editor creates the AST directly) and the Intentional Domain Workbench [12], which uses projectional editing even for languages whose concrete syntax looks textual.

Work to be done

More architectural features. Obviously, more architectural features will be added to the ADSL toolkit over time. Based on the more recent customer projects there is already a set of additional features we would like to support.

Java Generator. Also, the Java API generator is not yet completely up-to-date with regards to the variability of the language itself. More work needs to be put into the generator.

openArchitectureWare 5 The facilities for composing Xtext artifacts are limited. For example, there is not much support for grammar modularization in oAW Xtext 4.3. The same is true for composition and modularization of constraint files or other oAW artifacts. As a consequence, we have to do all the variability on text level, using those //# and //> comments in textual artifacts.

In the upcoming oAW 5 framework, the facilities for modularizing oAW artifacts, especially Xtext grammars, will be far more sophisticated.

References

[1] Eclipse platform, www.eclipse.org

[2] Eclipse Modeling Framewok, eclipse.org/emf

[3] openArchitectureWare, openarchitectureware.org

[4] Pure Systems GnbH, pure::variants, http://www.pure-systems.com

[5] BigLever, Software Gears, http://biglever.com/

[6] xADL, http://www.isr.uci.edu/projects/xarchuci/

[7] ACME ADL, http://www-2.cs.cmu.edu/~acme/

[8] AADL http://www.aadl.info/

[9] Autosar, http://www.autosar.org/find02_07.php

[10] SDF, http://www.syntax-definition.org/

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

[12] Intentional Domain Workbench, http://intentsoft.com/technology/IS_OOPSLA_2006_paper.pdf

[13] http://Graphviz.org

[14] http://prefuse.org/

[15] Markus Völter, Iris Groher, Product Line Implementation using Aspect-Oriented and Model-Driven Software Development, SPLC 2007, or at http://www.voelter.de/data/pub/VoelterGroher_SPLEwithAOandMDD.pdf

[16] Völter, Kircher, Zdun: Remoting Patterns, Wiley 2004

Domain-Specific Modelling Language for Navigation Applications on S60 Mobile Phones

Janne Merilinna

VTT Technical Research Centre of Finland
P.O. Box 1000, 02044 Espoo, Finland
janne.merilinna@vtt.fi

Abstract

Domain-Specific Modelling Languages (DSML) provide an opportunity to have end-users at the centre of the software development process. Although end-users are seldom software developers, providing a language that both the end-users and software developers understand enables fluent communication between the stakeholders. In this paper, work in progress in the development of a DSML for navigation applications on positioning enabled S60 mobile phones is presented. The presented language enables the end-user to instantly experience the impact of changes in the models, by utilising a code generator that produces complete applications from the models. The architecture of both the supporting software framework and the generated applications are also discussed.

Categories and Subject Descriptors D.1.7 [**Programming Techniques**]: Visual Programming

General Terms Languages, Experimentation, Human Factors

Keywords code generation, end-user driven development

1. Introduction

Obtaining specifications for a software product directly from the end-users is worthwhile. In an optimal case, this enables the transforming of the end-users' will directly to a product. However, the end-users are seldom software developers themselves, thus the lack of a common language between the software developers and the end-users may become a barrier.

Domain-Specific Modelling Languages (DSML) can enable a fluent communication between the software developers and the end-users by providing a language easy enough to learn and understand [1]. This is achievable by providing a language that utilizes elements existing in the problem domain, instead of elements of a solution-space, which can be a stumbling block when using general-purpose modelling languages. By doing so, the end-users are already familiar with the language concepts, thus the learning curve is not too steep.

With code generation being a central process in the Domain-Specific Modelling (DSM), the possibility of transforming the models directly into a working application is feasible [1]. This enables end-users to instantly see the results of the modelling, thus enabling active participation in the requirements gathering and prototyping phases or even developing the software alone.

In this paper, we approach the end-user driven development with an experiment of developing a navigation applications product family [2] architecture for positioning-enabled S60 [3] mobile phones such as Nokia N95 [4]. The products of the family are not solely restricted to be composed of features selectable from a predefined list since a modelling language dedicated for the modelling of innovative navigation applications is also provided. The modelling language is supported by a code generator that generates complete code from the models.

The language is striven to be developed in such a way that even enables non-programmers, who would not otherwise be able to develop applications, to do so with the provided language. With the language, modellers are able to develop innovative navigation applications by utilizing map data provided by the OpenStreetMap[1] (OSM) [5] and are able to navigate both outdoors with GPS and indoors with the Database Correlation Method (DCM) [6].

This paper is structured as follows. First, the developed modelling language is illustrated by presenting a simple navigation application. Second, the implementation of the supporting software framework and the application architecture is presented. A discussion and conclusions close the paper.

2. Illustration of the DSML for Navigation Applications on S60 Mobile Phones

Fundamentally, navigation applications can be considered as applications that utilise positioning sensors, such as GPS, in order to show the location of the user on a map. Nevertheless, it is also common to have at least the following features:

- Zooming and panning of the map,
- Navigation, i.e. routing from source to destination, by utilising various routing criteria, and
- Browsing Point of Interests (POI), e.g. searching for the nearest bars, restaurants etc.

The other more advanced features can be considered to be composed from the above mentioned features.

Next in this section, the modelling language for navigation applications on positioning enabled S60 mobile phones (DSML for NavApp) is introduced by presenting an example application modelled with the developed language. Due to space limitations, not all of the language concepts can be presented here, but the example application should enable one to have an idea of the

[1] available under Creative Commons Attribution-ShareAlike 2.0

modelling language. Metaedit+ from Metacase [7] is utilized as a modelling environment.

2.1 Relation to the Existing S60 Language

Metaedit+ provides a set of example languages such as DSML for S60 [1, pp. 160-185]. This language includes a subset of elements provided by the Python for S60 (PyS60) [8]. The language enables the modelling of S60 applications almost in the WYSIWYG principle, thus providing a good starting point for the development of the DSML for NavApp.

Figure 1 represents the relation between the DSML for S60, DSML for NavApp, PyS60 and the S60 framework. As depicted, the DSML for NavApp also includes concepts of the existing language with additional concepts of its own.

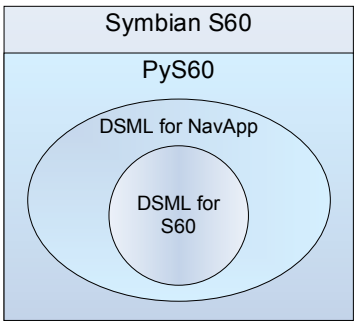


Figure 1. Relationships of the languages and frameworks.

2.2 A Simple Navigation Application as an Example

A simple navigation application is presented as an illustration of using the language. Figure 2 represents what will occur when the application starts. In this case, a pop-up dialog is shown first with three options. The application closes by choosing "Exit with the left sofkey, i.e. a button for accessing context-sensitive menus appearing at the bottom left of the screen of the mobile phone. If "About" is chosen, a note is shown and, as default behaviour after the note has disappeared, the application returns to the pop-up state. If "NavApp" is chosen, a navigation application called "NavApp" is launched. When the "NavApp" is closed, as default behaviour, the application returns to the pop-up state.

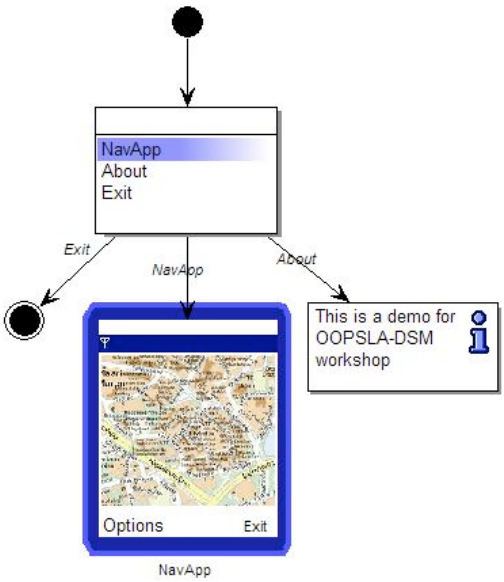


Figure 2. NavApp state machine.

Figure 3 represents the sub state machine of the NavApp, which is accessed from the NavApp entity in Figure 2 by decomposition. First, the menus and keyboard are defined with a "Menus" entity that provides a sub diagram where the definitions take place (see Figure 4). After the definitions, the application enters a loop where it is polled if the user is at the destination location. The destination can be defined by accessing one of the menus defined in Figure 4. When at the destination, a note is displayed to inform the user. The positioning ends after the note disappears.

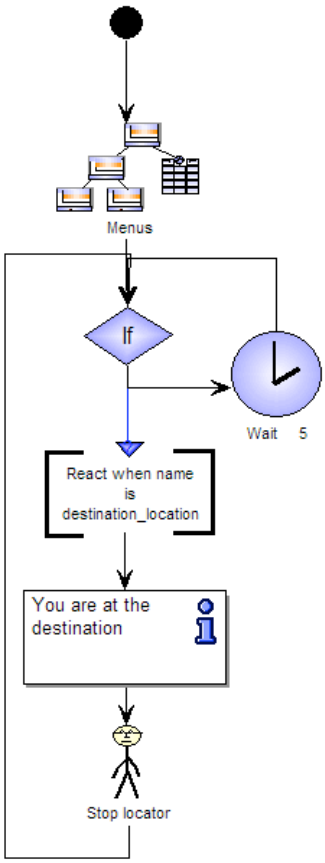


Figure 3. NavApp sub state machine.

Figure 4 represents definitions of the menus and the keyboard. First, zooming is attached to the <*+> and <#> keys followed by select menu definitions. By pressing the left softkey, a menu structure can be displayed where there is one parent node called "Navigate" having two child nodes, i.e. "Navigate to destination", and "Stop navigating". By choosing the latter, the navigation ends. By choosing the first mentioned, the user can type the name of the place where to navigate. After that, the route to the destination is computed from the current location and the navigation is started.

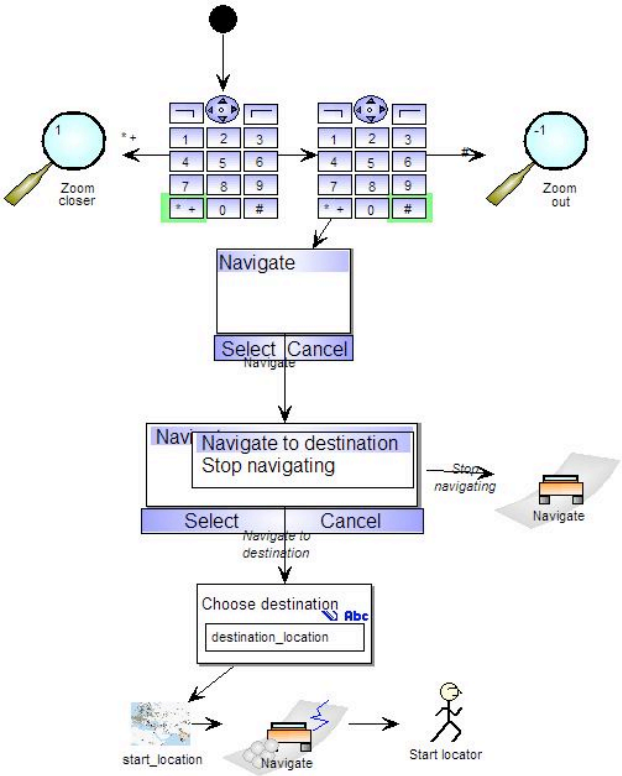


Figure 4. Menu and keyboard action definitions.

After the modelling, the code generator takes the models as an input and generates a Python source code on top of the supporting software framework. The generated application can then be installed into the phone. No additional manual source code writing is required.

3. Implementation of the NavApp

3.1 Architecture of the NavApp Framework

The basic architecture of the DSM can be considered to consist of three layers which are the software framework, the code generator and the metamodel. As illustrated in Figure 5, the metamodel provides all the rules on how to model applications. The code generator is responsible for taking modelled applications as an input and generates code from the models. It is common to have a software framework on top of the target platform, in order to make the code generation easier. [1]

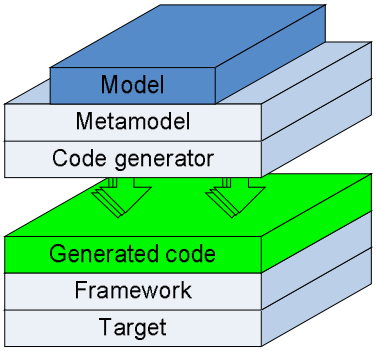


Figure 5. The basic architecture of DSM.

The primary driver for the NavApp framework is in the usability and extensibility of the framework. The interface for the framework is developed in such way that generating source code for the applications is straightforward. The framework encapsulates functions in a way that the generated applications mainly consist of function calls to the framework, in addition to utilizing the PyS60 framework. These requirements are materialised as a façade that hides all unnecessary details. Figure 6 represents all the relevant parts of the NavApp framework architecture.

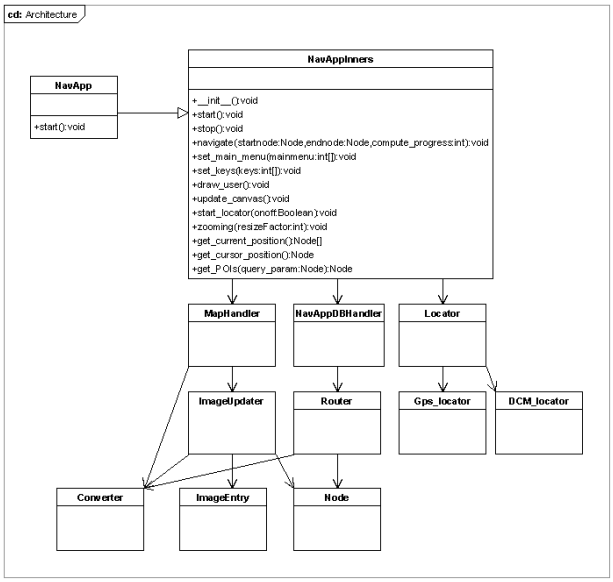


Figure 6. NavApp framework architecture.

The NavApp framework architecture is divided into four fundamental components:

- NavAppInners functions as a façade for the rest of the framework. It is the only NavApp-specific class that is accessed from the generated code.
- MapHandler is responsible for creating and handling the map, showing a route when navigating, and the other relevant visualization actions related to the map. In addition, MapHandler provides a set of callback functions to be used by the other components for displaying info on the map. Currently, pre-rendered images provided by the OSM sub-project, Tiles@home [9], are utilized as a map instead of rendering the map from the OSM data.
- NavAppDBHandler is responsible for handling the navigation data, computing route and also conducting queries directly to the OSM data for POIs etc. In addition, NavAppDBHandler provides facilities for storing additional data to the database.
- Locator is responsible for positioning. Positioning is performed by utilizing two of the most widely used positioning technologies, GPS and WLAN. WLAN positioning is based on an inbuilt DCM [6].

3.2 Architecture of the NavApp Applications

The code generator for the DSML for S60 [1, pp. 160-185] generates applications running on a state machine. Each entity is generated as a state realized by a function, where the states maintain a reference to the next state. In order to incorporate code generated from the DSML for NavApp, the generated code has to conform to that state machine. Therefore, the entity that initializes NavApp is generated similarly as any other state in the state machine generated from the DSML for S60. Thus, the introduction of the NavApp entity into the existing code generator and the language does not have any particular impact on the other entities.

The actual implementation entity of the NavApp which the initialization function calls is generated as a class that inherits the NavAppInners (see Figure 6). The NavApp maintains its own internal state. Similar to the code generated from DSML for S60 models, NavApp internal states are generated as states that are realized as functions, where all the functions maintain a reference to the next state.

Sub state machine and keyboard and menu definitions are generated differently. Whilst the sub state machine is generated as states running on the state machine, keyboard and menu definitions are generated as an encapsulated state in the sub state machine.

4. Discussion

The current version of the language is still immature and not as polished as possible, thus it requires one to become familiar with it. Therefore, the consideration of end-user driven development is still a matter of debate.

Currently, the language and framework are in active development. We are adding, among others, a possibility of utilizing an external positioning server in order to enable multi-person positioning and to enable the development of multi-user position-based games and to bring a social media dimension to the navigation applications.

5. Conclusions

In this paper, the work in progress in the development of the DSML for NavApp and its supporting framework is presented. The language is developed in a way that could enable non-programmer end-users to actively participate in the development of navigation applications or to develop applications completely by themselves. By utilizing the presented language, end-users can instantly experience the impact of changes in the model as the provided code generator enables complete code generation from the models.

[1] Kelly, S. and Tolvanen, J-P, "Domain-Specific Modeling: Enabling full code generation", John Wiley & Sons, ISBN 978-0-0470-03666, 2008, 427 p.

[2] J. Bosch, "Design & Use of Software Architectures: Adopting and Evolving a Product-Line Approach", Addison-Wesley, 2000.

[3] S60.com, URL: http://www.s60.com/life [Visited at 9.7.2008]

[4] Nokia.com, Nokia N95, URL: http://www.nseries.com/products/n95/#l=products,n95 [Visited at 9.7.2008]

[5] OpenSreetMap, The Free Wiki World Map, URL: http://www.openstreetmap.org/ [Visited at 9.7.2008]

[6] Kemppi, P. Nousiainen, S, "Database Correlation Method for GSM Location", Vehicular Technology Conference, VTC 2000, IEEE VTS 53rd, 2001.

[7] Metaedit+ WorkBench, URL: http://www.metacase.com/mwb/ [Visited at 9.7.2008]

[8] Nokia Research Centre, Python for S60, URL: http://opensource.nokia.com/projects/pythonfors60/ [Visited at 9.7.2008].

[9] Tiles@home, URL: http://tah.openstreetmap.org/ [Visited at 9.7.2008]