ACM/IEEE 18th International Conference on
Model Driven Engineering Languages and Systems

September 29, 2015 – Ottawa (Canada)

Flexible Model Driven Engineering Proceedings (FlexMDE 2015)

Davide Di Ruscio, Juan de Lara, Alfonso Pierantonio (Eds.)

Editors' addresses:

Davide Di Ruscio
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica
Università degli Studi dell'Aquila (Italy)

Juan de Lara
Escuela Politécnica Superior
Departamento de Ingeniería Informática
Universidad Autónoma de Madrid (Spain)

Alfonso Pierantonio
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica
Università degli Studi dell'Aquila (Italy)

Preface

Increasingly, models are starting to become commonplace and Model Driven Engineering (MDE) is gaining acceptance in many domains including automotive, aerospace, railways, telecommunications, business applications, and financial organizations. Over the last years, several modeling platforms have been developed to simplify and automate many steps of MDE processes. However, still several challenges have to be solved in order to enable a wider adoption of MDE technologies. One of the most important impediments in adopting MDE tools is related to the reduced flexibility of existing modeling platforms that do not permit to relax or enforce their rigidity depending on the stages of the applied development process. For instance, EMF does not permit to enter models which are not conforming to a metamodel: on one hand it allows only valid models to be defined, on the other hand it makes the corresponding pragmatics more difficult. In this respect, there is a wide range of equally useful artefacts between the following extremes:

- diagrams informally sketched on paper with a pencil
- models entered in a given format into a generic modeling platform, e.g., Ecore/EMF

At the moment, modeling platforms encompasses only the latter possibility. However, while depending on the stage of the process it makes sense to start with something closer to the former (to promote communication among stakeholders) to eventually end up with the latter (to allow automatic model processing and code generation). Thus, we are interested in exploring the possible forms of flexibility that are required when applying MDE processes, ranging from agile ways to develop modeling artifacts and languages to their flexible application in concrete domains.

Flexibility is also needed to enable wider possibilities for reusing MDE artefacts, like model transformations and code generators. In particular, to deal with the growing complexity of software systems, it is necessary to enforce consistent reuse and leverage the interconnection of the modeling artifacts that are produced and consumed during the different phases of the applied development processes. In such contexts, modularization mechanisms have to be devised in order to enable the development of complex modeling artifacts from smaller ones, which are easier to process and reuse.

The first edition of the Flexible Model Driven Engineering workshop (FlexMDE) – previously known as Extreme Modeling Workshop (XM) - has been co-located with the ACM/IEEE 18th International Conference on Model Driven Engineering Languages & Systems (MODELS 2015). It provided a forum for researchers and practitioners where different forms of agility have been discussed as demonstrated by the technical program, including agile development of domains specific languages, agile requirements engineering, and fast development of data-intensive Web applications. The program included also a keynote by Prof. Colin Atkinson (University of Mannheim) entitled "Maximizing the Amount of Information Not Modelled in MDE". As part of the workshop, an informal tool demonstrations session has been also organized consisting of 5 tools aiming at adding flexibility to MDE at different extents.

Many people contributed to the success of FlexMDE 2015. We would like to truly acknowledge the work of all Program Committee members, and reviewers for the timely delivery of reviews and constructive discussions given the very tight review schedule. Finally, we would like to thank the authors, without them the workshop simply would not exist.

October 2015

Davide Di Ruscio
Juan de Lara
Alfonso Pierantonio

Organizers

Davide Di Ruscio (co-chair)	Università degli Studi dell'Aquila (Italy)
Juan De Lara (co-chair)	Universidad Autonoma de Madrid (Spain)
Alfonso Pierantonio (co-chair)	Università degli Studi dell'Aquila (Italy)

Program Committee

Colin Atkinson	University of Mannheim (Germany)
Paolo Bottoni	Sapienza University of Rome (Italy)
Antonio Cicchetti	Maalardalen University (Sweden)
Tony Clark	Middlesex University (UK)
Jeff Gray	University of Alabama (USA)
Robert Hirschfeld	Hasso-Plattner-Institut (Germany)
Dimitris Kolovos	University of York (UK)
Philipp Kutter	Montages AG (Switzerland)
Mark Minas	Universität der Bundeswehr München (Germany)
Rick Salay	University of Toronto (Canada)
Jesus Sanchez Cuadrado	Universidad Autonoma de Madrid (Spain)
Bran Selic	Malina Software Corp. (Canada)
Jim Steel	University of Queensland (Australia)
Vadim Zaytsev	Universiteit van Amsterdam (NL)
Steffen Zschaler	King's College London (UK)

Table of Contents

Keynote

Maximizing the Amount of Information Not Modelled in MDE

Colin Atkinson
University of Mannheim

Ideally, a domain-specific modeling languages should allow domain experts to define requirements and solutions with the minimal necessary model content. Any model content that is not strictly necessary for expressing the desired message is essentially “accidental complexity” and reduces the generality and “flexibility” of the model. Therefore, in the same way that agile methods aim to “maximize the amount of work not done” in software engineering, model-driven development should aim to maximize the amount of information not modeled in the fulfillment of modeling goals. This requires the use of languages that are not only optimized to express concepts in the domain of discourse but also possess clear inference and frame rules allowing modelers to infer information that is not explicitly expressed. Examples include “world assumptions” (e.g. open or closed), derivation mechanisms (e.g. inheritance), “completion” statements and elision notations. In this talk Colin Atkinson will clarify what it means to maximizing the amount of information not modelled in MDE, suggest some concrete principles and mechanism for achieving this goal, and explore what consequence it can have for model flexibility.

Colin Atkinson *is leader of the Software Engineering Group at the University of Mannheim (Germany). Before that he held a joint position as a professor at the University of Kaiserslautern and project leader at the affiliated Fraunhofer Institute for Experimental Software Engineering. From 1991 until 1997 he was an Assistant Professor of Software Engineering at the University of Houston Clear Lake. His research interests are focused on the use of model-driven and component based approaches in the development of dependable computing systems. He received a Ph.D. and M.Sc. in computer science from Imperial College, London, in 1990 and 1985 respectively, and received his B.Sc. in Mathematical Physics from the University of Nottingham 1983.*

MUTANT: Model-Driven Unit Testing using ASCII-art as Notational Text

Daniel Strüber, Felix Rieger, Gabriele Taentzer

Philipps-Universität Marburg, Germany,
`{strueber, riegerf, taentzer}@informatik.uni-marburg.de`

Abstract. There are two established strategies to create test models: Textual notations require developers to mentally reconstruct the involved graph structures *ad hoc*; maintenance effort and time are increased. Existing visual notations compel developers to switch frequently between different editors; keeping the resulting test artifacts synchronized is complicated by insufficient tool support. In this paper, we propose to specify test models using ASCII-art – a text-based visual notation used by developers in their everyday practices. To outline our vision, we employ a motivating example and discuss challenges such as editing, collaboration, and scalability. The discussion sheds a promising light on the new idea, notably for its usefulness in collaborative and reuse-intensive settings.

1 Introduction

Model-Driven Engineering (MDE) emerged as a paradigm to combat complexity in software engineering through the use of visual notations. While its basic premise appears promising, MDE has not found broad adoption as a mainstream software development methodology, and a major cause for this failure is often seen in inadequate tool support [1]. In particular, Batory et al. found the graphical tools in a widely-known modeling framework unsuitable for teaching MDE to students, considering them unappealing and unintuitive [2].

This lack of enthusiasm is seconded by practitioners. In a recent online discussion, an industry participant observed that *"the graphical tools are bulky, lack often some features like (smart) versioning, merging, collaborating, good integration into the overall development tool chain. [...] Working with mouse, property dialogs, popup windows never gains the speed of a well configured source IDE."*[1]

In this work, we aim to provide the benefits of visual models without inducing any of these tool-related problems. The idea is to specify models using *ASCII-art*, a text-based visual notation used by developers in their everyday practices [3]. ASCII-art is widely used in specification documents, such as Internet RFCs [4]. We will explore this idea in a context where it appears particularly well-suited: Simplifying *unit testing*, a cornerstone in software quality assurance [5].

[1] *http://modeling-languages.com/failed-convince-students-benefits-code-generation* User *Det*, 2015-02-11

Often, a system under test takes as input a data structure with an intuitive visual representation: Consider (i) a routing algorithm finding shortest paths in a graph of towns, (ii) a model transformation deriving database schemas from class models, or (iii) a social network site proposing new friends based on a social graph. Adopting MDE terminology, we will call such data structures *models*.

There are two established strategies to create test models: The first is textual specification using APIs or dedicated DSLs. However, textual notations are not suited to capture the visual nature of the involved models. It has been reported that the use of visual models respecting certain layouting criteria has a positive effect on cognitive load [6]. Hence, the second option is to construct test models using visual tools. Yet this results in a complicated process, involving repeated switching between different editors and keeping test models and code synchronized – challenging tasks, given the lack of production-ready collaboration tools.

In this paper, we propose *model-driven unit testing using ASCII-art as notational text* (MUTANT): Unit tests are annotated with test models, using a visual notation based on ASCII characters. The main component of the approach is a *model compiler* that derives models from these annotations and provides these models to the test framework. By adding it to the build script, this compiler can be integrated into any given IDE. We put forth the following design rationale:

I. Diffing and merging annotated code, crucial tasks in versioning and collaboration, is supported by mature tools included in state-of-the-art IDEs.

II. Collaborators and project stakeholders, such as managers and customer-side developers, can view test models without requiring any tool setup.

III. The notation is not bound to a specific modeling platform. Alternative platforms can be supported by providing separate model compilers.

IV. Creating new test models from existing ones boils down to copying text. Sec. 2 reports on a daunting experience of performing this task using visual tools.

The rest of this paper is structured as follows: In Sec. 2, we introduce the approach by example. In Secs. 3 and 4, we discuss challenges and an implementation prototype. We discuss related work and conclude in Secs. 5 and 6.

2 Motivating Example

Consider the following example to compare the new approach against the established approaches to test model specification. The system under test is an implementation of the *pull up attribute* refactoring [7] for class models: If two classes have the same superclass and attributes of the same type and name, the attributes are moved to the superclass. The unit tests in this example involve a part where a test model is provided, a part where the refactoring is applied, and assertions to check if the attribute was pulled up.

Textual specification. Fig. 1 presents a unit test for the refactoring based on the Eclipse Modeling Framework (EMF), a modeling platform often used to create and persist models [8]. In lines 4-12, the test model is created: A *package* acting as overarching container, *classes*, and their *attributes* are created. In lines 13-14, `person` is set as common superclass for the `professor` and `student` classes. In lines 16-19, the refactoring is applied and assertions are checked.

```java
public class RefactoringTest extends UnitTest {
  @Test
  public void testRefactoring() {
    EFactory fact = EcoreFactory.eINSTANCE;
    EPackage pkg = fact.createEPackage();
    EClass person = fact.createEClass(pkg);
    EClass professor = fact.createEClass(pkg);
    EClass student = fact.createEClass(pkg);
    EAttribute attr1 = fact.createEAttribute(
      "name", String.class, professor);
    EAttribute attr2 = fact.createEAttribute(
      "name", String.class, student);
    professor.setSuperClass(person);
    student.setSuperClass(person);

    assertTrue(person.getAttributes().size()==0);
    PullUpRefactoring refac = new
        PullUpRefactoring(pkg);
    refac.execute();
    assertTrue(person.getAttributes().size()==1);
  }
}
```

Fig. 1. Textual test model specification.

```java
public class RefactoringTest extends UnitTest {
  @Test
  public void testRefactoring() {
    EPackage pkg = load("/test1/Test.ecore");
    EClass person = pkg.getEClass("Person");

    assertTrue(person.getAttributes().size()==0);
    PullUpRefactoring refac = new
        PullUpRefactoring(pkg);
    refac.execute();
    assertTrue(person.getAttributes().size()==1);
  }
}
```

Fig. 2. Loading a visually specified test model.

This approach has two advantages: Test model and code are maintained at the same place, eliminating the need for context and tool switching. Since the input model is specified textually, it can be easily diffed and merged, facilitating collaboration. However, the easy tractability of the code comes at a price: The specification does not reflect the graph structure of the input model. Developers have to reconstruct the graph in their minds, adding a level of complexity to the understanding process and increasing maintenance effort and time.

External visual specification. Fig. 2 shows an equivalent unit test where the test model, created using a visual editor, is loaded from the file system.

This solution has two benefits: The input model is specified visually, and the test code remains clean and simple. On the downside, to understand and maintain tests, developers are forced to switch between input models and tests. Furthermore, the absence of the actual test model in this presentation reflects the situation collaborating developers might find themselves in: The committing developer might have used an incompatible version of the visual tool, or have forgotten to include the test model in the commit. From our own experience, we report on another serious drawback: To achieve a good test coverage, it is reasonable to reuse test models by copying and modifying them. In EMF, this is a complicated process: Models and their layout information are distributed over several files, using hard-coded references that have to be adjusted manually.

MUTANT. Fig. 3 exemplifies test specification using the proposed approach. The test model is provided in the Javadoc accompanying the test method, using a custom `@InputModel` parameter. In the employed syntax, boxes indicate classes. The generalization relation is indicated by arrows: The character `A` resembles a closed arrowhead. We provide a model compiler to parse such annotations and derive models, making them accessible through a dedicated API, invoked in line 18. To specify output models, the parameter `@OutputModel` can be used.

```
 1 public class RefactoringTest extends UnitTest {
 2   @Test
 3   /** @InputModel EPackage pkg =
 4 
 5                    +------------+
 6                    |   Person   |
 7                    +------------+
 8                        A   A
 9              .-------'   '-------.
10              |                   |
11        +--------------+  +--------------+
12        | Professor    |  | Student      |
13        |--------------|  |--------------|
14        | name: String |  | name: String |
15        +--------------+  +--------------+
16    */
17   public void testRefactoring() {
18     EPackage pkg = Mutant.getPackage("pkg");
19     EClass person = pkg.getEClass("Person");
20 
21     assertTrue(person.getAttributes().size()==0);
22     PullUpRefactoring refac = new
         PullUpRefactoring(pkg);
23     refac.execute();
24     assertTrue(person.getAttributes().size()==1);
25   }
26 }
```

Fig. 3. Specifying a test model using ASCII-art.

The outlined solution combines the advantages of both previous solutions without suffering from any of their drawbacks: It establishes locality and use of text-based collaboration tools while maintaining an intuitive, graphical layout, indicating the layout of the known graphical model notation.

3 Challenges and Vision

In this section, we outline the full vision of the proposed approach, highlighting its strengths, challenges, and strategies to tackle these challenges.

Editing. An important benefit of ASCII-art over non-textual graphical representations is that it can be created, edited and viewed using any text editor. However, not all text editors are recommendable for all of these tasks. Depending on the editor at hand, the editing process can be cumbersome: For instance, moving a box horizontally may require to add whitespace in successive lines.

Instead, our approach targets state-of-the-art code IDEs. These IDEs come with powerful text editors: Eclipse, IntelliJ and Visual Studio provide dedicated modes allowing to manipulate successive lines by a single keystroke.[2] Capabilities to add, insert or remove characters in a column-wise fashion and to select or manipulate *boxes* facilitate ASCII-art editing noticeably. To accommodate larger editing steps, we propose the use of specialized text editors, providing convenient features such as box drawing and freehand erasers.[3] For viewing and minor edits, the IDE code editor remains the designated tool – considerably reducing the need for context switching imposed by processes involving graphical editors.

Collaboration. We base our approach on the claim that employing ASCII-art enables the use of the mature collaboration tools found in present-day IDEs. More specifically, we maintain that the existing line-based *diff* and *merge* tools are sufficient in most cases. Yet, we identify two caveats:

First, text differencing considers just the syntax of the involved models and is oblivious to their semantics. As a consequence, syntactically different models are reported as different, even if they are semantically equivalent. This is analogous to code diffs, where two semantically equivalent lines of code may be highlighted as different. Second, as the only problematic case for merging, we identify merge conflicts concerning the same line of code, which have to be resolved by hand – again, a situation that is accepted in source code editing.

Scalability. The proposed idea is particularly suited for the creation of unit tests since the models involved in unit testing tend to be relatively small: Unit tests usually represent *edge cases* reflecting the core of a critical scenario. However, particular edge cases may demand large models. To address these cases, we must account for the inherent limitations of text-based notations: Zooming is not available. Space may be limited to 80 or 100 characters per line.[4]

[2] Eclipse: *block selection mode*, IntelliJ and Visual Studio: *column mode.*

[3] e.g. http://asciiflow.com/, http://www.jave.de/, http://emacswiki.org/emacs/ArtistMode. Retrieved on 2015-08-31.

[4] The example model in Fig. 3 stretches over 33 characters of horizontal dimension.

In order to address the space limitation, we provide adjustments that help to increase notational compactness. To compact multiple objects of the same type, nodes can carry a multiplicity, indicated by the character `n`. To compact multiple links of the same type, edges can be multi-edges, i.e., have multiple source or targets. To remove visual clutter in models with many edges – a potential obstacle to developer performance [9] –, we introduce *abbreviated edges*, a concept inspired by net labels in ECAD software[5]. The example model in Fig. 4 shows a school with ten teachers, nine of them named *Mary*, one named *Ed*.

```
     +------------+
     | :School    |#--{teachers}--[c]
     +------------+
+----------------+         +-------------+
| :Teacher [n=9] |  [c]--->| :Teacher    |
|----------------|         |-------------|
| name = "Mary"  |<---[c]  | name = "Ed" |
+----------------+         +-------------+
```

Fig. 4. Test model with multi-node and abbreviated multi-edge.

In the future, we aim to empirically investigate the scalability of the proposed approach, including the effect of these mitigation strategies. In any case, it might be argued that the advantage of unlimited space and zooming in graphical tools is debatable in the first place: It has been observed that diagrams exceeding a certain size can present an obstacle rather than an aid to comprehension [10].

Scope. To make the approach broadly applicable in various application domains, we aim to provide support for different modeling languages. All modeling languages can be supported by considering their *abstract syntax* – a representation based on boxes and arrows, exemplified in Fig. 4. Still, developers may prefer the known *concrete syntax* (CS) notations. The example in Fig. 3 presents a CS-based notation for class models. To support extensibility for arbitrary languages, we propose a framework approach, allowing customization for new languages by defining visual tokens and their mapping to the underlying meta-model.

Syntax errors. In present-day visual and textual editors, developers are supported by syntax checks, allowing them to discover specification errors early, during editing rather than at runtime. Feedback on syntax errors is part of our basic approach: Its central component, a model compiler, is able to return specific error messages in case that syntax errors are found. These error messages are forwarded to the IDE by means of the build script invoking the compiler.

Conversion. It is desirable to enable an easy transition between external specifications and the proposed notation. Sources of external specification may include regular models in custom layout formats, PowerPoint and Visio documents, and even hand-drawn sketches photographed using smartphone cameras.

[5] e.g., KiCad, http://tinyurl.com/kicad-labels/. Retrieved on 2015-08-31.

As a promising approach to address this sophisticated task, we aim to use an ASCII-art generation heuristics that accounts for line structures found inside an input graphic [11]. Postprocessing is required to ensure that valid instances of the proposed syntax are created. Furthermore, if the existing test models were generated automatically, they might not have a layout in the first place or be too large for a visualization. To support them, we intend to apply state-of-the-art layouting [12, 13] and splitting tools [14–16].

4 Implementation Prototype

We provide a prototypical implementation for the proposed approach. The main component of the implementation is a model compiler that can be plugged into any given IDE by adding an entry to the build configuration or script. During compilation of annotated test classes, the compiler parses the contained ASCII-art annotations to create models which are made available to the test framework by means of a dedicated API. The implementation and instructions for plugging it into Eclipse are found at `https://github.com/frieger/mutant-ascii`.

We support class models through a dedicated concrete syntax and arbitrary modeling languages through abstract syntax. The parser first identifies boxes and extracts contained information on names and types. Then, it identifies arrow-heads, following the adjacent edges until a box or abbreviated edge label is hit. It detects and follows remaining abbreviated edges, connecting those with identical labels and converting multi-edges to multiple single edges. The information collected on objects and their relations serves as input to a model builder.

We initially planned to use Java annotations for specifying test models, which proved infeasible since multi-line String literals are not supported in Java. Instead, as shown in Fig. 3, we embed models in Javadoc. Conceptually, Javadoc is a suited place: The input and output of the code under test are documented.

5 Related Work

5.1 Model-Driven Testing

The proposed approach can be considered as a novel incarnation of model-driven testing (MDT). MDT places models as key artifacts in testing. In earlier MDT approaches, the aim was to derive test models using abstract specifications such as *coverage criteria* [17], *dedicated profiles* [18], and *visual contracts* [19]. While automated generation of test cases is a desirable and well-studied goal [20], the trade-off is a significant initial cost in adopting the associated methods and tools. From an enterprise perspective, this poses a considerable risk, notably when one takes into account that the approaches are not tailored to all relevant tasks: For instance, the approaches require to specify behavior using rules or sequence diagrams, focusing on changes of object structures and neglecting algorithmic functionality such as graph routing. Specifications are translated to plain test cases that may expose the indicated drawbacks.

Unlike previous approaches to MDT that focus on the generation of test models, the new approach takes an agile stance, allowing rapid test model creation: Developers specify test models directly, using a notation designed for easy reuse and collaboration. This allows focusing on the intuitive process of identifying *edge cases*. To our knowledge, our approach is the first to represent models in a dedicated notation to facilitate testing. To still reap the benefits of the existing MDT approaches, we aim to provide converters for the derived test models.

5.2 ASCII-art Notations in Software Engineering

Several software engineering problems have been tackled using ASCII-art. *TextTest* [21] is a tool for graphical user interface (GUI) testing based on the capture-replay paradigm: Developers interact with the GUI under test. After each interaction, a GUI snapshot is saved using ASCII-art, enabling automated regression tests. This approach is complementary to ours: Capture-replay is only available for GUIs, while our approach targets at the broad class of functionality tests that involve *models*. Furthermore, documenting each interaction leads to many text artifacts, not promoting locality and easy reuse. Another complementary approach [22] uses an ASCII-based model notation for code generation. The authors mention converters from models to ASCII-art and back; however, they do not explicate their realization strategy. They consider class models. In [23], diagram parsing is used to recover grammars for existing programming languages from reference manuals. The authors discuss an interesting solution based on *attributed multiset grammars* [24]. [25] proposes a context-free grammar for ASCII-art tables as found in network protocol RFCs.

5.3 Visualizations in Textual IDEs

The lack of visual expressiveness associated with textual notations has motivated work on visualization in code IDEs. The *JetBrains MPS* language workbench [26] supports a form of integrated textual and visual editing: Language developers can define custom box-and-arrow type diagram views that are embedded into source code editors. Such embedded views mitigate several of the problems of purely visual or textual editing, such as comprehension effort and context switching.

On the downside, they only facilitate the editing process. Diffing and merging models remains a challenge. Moreover, this approach is coupled to a specific IDE, which raises multiple problems: Developers are forced to use this IDE, which is undesirable if a particular preferred IDE exists in their domain. Besides, as in any new and experimental IDE, the embedded editors may show some of the issues reported for graphical tools. Finally, specific IDEs come with an increased business risk: It is not guaranteed that support is continued in the future. In contrast, our approach offers a drop-in solution designed to support arbitrary IDEs and their mature versioning and collaboration tools. To combine the benefits of both approaches, we consider customizing MPS to use its embedded views as front-end editors for ASCII-art model representations.

mbeddr [27], an extension of JetBrains MPS targeted at embedded software development, provides built-in visualizations for state machines. The *Xtext* language workbench [28] allows to visualize instances of textual DSLs using the ZEST visualization library [29]. Both approaches provide read-only visualizations, while the embedded views in MPS are also editable.

5.4 Tool Reuse

Our premise of using production-ready tools rather than experimental ones designed for specific purposes is inspired by recent work on usability-oriented MDE tools. The *Visual Model Transformation Language* (VMTL) [30] allows to specify model transformations using regular model editors. Similar to the new approach, VMTL uses annotations to enable the reuse of existing technology: In VMTL, models are annotated to specify transformation rules. In this work, test code is annotated to specify test models. However, while VMTL allows to reuse model editors, the current work reflects our experience that these editors are not well-suited for test model creation – an issue we avoid by using textual IDEs.

6 Conclusion and Future Work

Tests are of paramount importance in software engineering. We target the challenge of model-driven unit testing: Instead of using external editors to view and edit test models, we embed the models in the Javadoc comments accompanying the test cases. The approach is text-based and does not modify the programming language's syntax, allowing to use existing editing, versioning, and collaboration tools. The text-based visual syntax is designed to resemble the well-known graphical notations while allowing to reduce visual clutter. As in visual tools, model elements are aligned freely, supporting comprehension through spatial clues.

We address a set of challenges and solution ideas that we aim to investigate more deeply in the future. These challenges include the development of converters from external specifications to ASCII-art, the development of a framework to support multiple modeling languages through their concrete syntax, and the empirical investigation of the approach's scalability and general usefulness. Tackling these challenges will lead to a set of domain- and IDE-independent tools enabling developers to write tests more easily, combining the benefits of Test-Driven Development and Model-Driven Engineering.

References

1. J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, "Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem?" in *Model-Driven Engineering Languages and Systems*. Springer, 2013, pp. 1–17.
2. D. Batory, E. Latimer, and M. Azanza, "Teaching Model Driven Engineering from a Relational Database Perspective," in *Model-Driven Engineering Languages and Systems*. Springer, 2013, pp. 121–137.
3. M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko, "Let's Go to the Whiteboard: How and Why Software Developers use Drawings," in *Conf. on Human Factors in Computing Systems*. ACM, 2007, pp. 557–566.

4. L. Zhu, V. Chen, J. Malyar, S. Das, and P. McCann, "RFC 7545: Protocol to Access White-Space (PAWS) Databases," 2015.
5. G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing.* John Wiley & Sons, 2011.
6. H. Störrle, "On the Impact of Layout Quality to Understanding UML Diagrams," in *Visual Lang. and Human-Centric Comp.* IEEE, 2011, pp. 135–142.
7. M. Fowler, *Refactoring.* Addison Wesley, 2002.
8. D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework.* Pearson Education, 2008.
9. D. Whitney and D. M. Levi, "Visual Crowding: A Fundamental Limit on Conscious Perception and Object Recognition," *Trends in cognitive sciences*, vol. 15, no. 4, pp. 160–168, 2011.
10. H. Störrle, "On the Impact of Layout Quality to Understanding UML Diagrams: Size Matters," in *Model-Driven Engineering Languages and Systems.* Springer, 2014, pp. 518–534.
11. X. Xu, L. Zhang, and T.-T. Wong, "Structure-based ASCII art," *ACM Transactions on Graphics (TOG)*, vol. 29, no. 4, pp. (52) 1–10, 2010.
12. M. Spönemann, *Graph Layout Support for Model-Driven Engineering.* PhD diss., Uni Kiel, 2015.
13. S. Maier and M. Minas, "A Pattern-based Approach for Initial Diagram Layout," *Electronic Communications of the EASST*, vol. 58, 2013.
14. D. Strüber, M. Selter, and G. Taentzer, "Tool support for clustering large metamodels," in *Proceedings of the Workshop on Scalability in Model Driven Engineering.* ACM, 2013, p. 7.
15. D. Strüber, J. Rubin, G. Taentzer, and M. Chechik, "Splitting Models using Information Retrieval and Model Crawling Techniques," *Fundamental Approaches to Software Engineering*, pp. 47–62, 2014.
16. D. Strüber, M. Lukaszczyk, and G. Taentzer, "Tool Support for Model Splitting using Information Retrieval and Model Crawling Techniques," in *Proceedings of the Workshop on Scalability in Model Driven Engineering. ACM*, 2014.
17. R. Heckel and M. Lohmann, "Towards Model-Driven Testing," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 6, pp. 33–43, 2003.
18. P. Baker, Z. R. Dai, J. Grabowski, I. Schieferdecker, and C. Williams, *Model-Driven Testing: Using the UML Testing Profile.* Springer, 2007.
19. G. Engels, B. Güldali, and M. Lohmann, "Towards Model-Driven Unit Testing," in *Models in Software Engineering.* Springer, 2007, pp. 182–192.
20. S. Anand, E. Burke, T. Chen, J. Clark, M. Cohen, W. Grieskamp, M. Harman, M. Harrold, and P. McMinn, "An Orchestrated Survey of Methodologies for Automated Software Test Case Generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.
21. E. Bache and G. Bache, "Specification by Example with GUI Tests-How Could That Work?" in *Agile Processes in Software Engineering and Extreme Programming.* Springer, 2014, pp. 320–326.
22. H. Washizaki, M. Akimoto, A. Hasebe, A. Kubo, and Y. Fukazawa, "TCD: A text-based UML class diagram notation and its model converters," in *Advances in Software Engineering.* Springer, 2010, pp. 296–302.
23. R. Lämmel and C. Verhoef, "Semi-automatic Grammar Recovery," *Software: Practice and Experience*, vol. 31, no. 15, pp. 1395–1438, 2001.
24. S.-K. Chang, "Picture Processing Grammar and its Applications," *Information Sciences*, vol. 3, no. 2, pp. 121–148, 1971.
25. A. Kay, D. Ingalls, Y. Ohshima, I. Piumarta, and A. Raab, "Steps toward the Reinvention of Programming," Technical report, National Science Foundation, Tech. Rep., 2006.
26. M. Voelter and K. Solomatov, "Language modularization and composition with projectional language workbenches illustrated with MPS," *Software Language Engineering*, vol. 16, 2010.
27. M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb, "mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems," in *C. on Systems, Progr., and Apps.* ACM, 2012, pp. 121–140.
28. M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in *ACM International Conf. Object-Oriented Programming Systems Languages and Applications Companion.* ACM, 2010, pp. 307–309.
29. R. I. Bull, *Model Driven Visualization: Towards a Model Driven Engineering Approach for Information Visualization.* PhD diss., University of Victoria, 2008.
30. V. Acretoaie, H. Störrle, and D. Strüber, "Transparent Model Transformation: Turning Your Favourite Model Editor into a Transformation Tool," in *International Conf. on Model Transformations.* Springer, 2015, pp. 121–130.

Agile bottom-up development of domain-specific IDEs for model-driven development*

Steffen Vaupel, Daniel Strüber, Felix Rieger, Gabriele Taentzer

Philipps-Universität Marburg, Germany
`{svaupel,strueber,riegerf,taentzer}@informatik.uni-marburg.de`

Abstract. Diminishing time-to-market and rapidly evolving technology stacks stretch traditional software development methods to their limits. In this paper, we propose a novel process for bottom-up development of domain-specific IDEs based on agile principles. It aims to enable a fine-grained co-evolution of domain-specific modeling languages (DSMLs) and their model editors and code generators. We illustrate our approach by iteratively developing an IDE for model-driven development of mobile applications. As a key success factor for continuous DSML development, we determine the automated deduction of migration scripts for all dependent artifacts of a DSML evolution step.

1 Introduction

Vastly increasing numbers of applications and users make the development of mobile applications one of the most important fields in software engineering. In this field, short time-to-market, differing platforms and rapidly emerging technologies stretch traditional software development methodologies to their limits. A viable research direction to tackle these challenges involves the combination of model-driven development (MDD) [1] and agile software development [2]: Aligning the domain-specific, platform-independent abstractions provided by MDD with agile principles such as *quick response to change* and *early delivery* promises a high potential to handle the requirements of rapidly evolving software domains.

Experience has shown that model-driven and agile practices complement each other well during the development of individual applications [3,4]. In these scenarios, a static modeling language was assumed; the evolution of this language was not considered. Yet in the reality of rapidly evolving software domains, the evolution of domain-specific modeling languages (DSMLs) has become an unavoidable fact. A major challenge is the co-evolution of the enabling technologies for DSMLs – in particular, their IDEs. Domain-specific IDEs include model editors and code generators. The development of these components is facilitated by a wealth of meta-tools: GMF [5], Sirius [6] and Xtext [7] for editor, Xtend [7] and EGL [8] for generator development.

* This work was partially funded by LOEWE HA project no. 355/12-45 (State Offensive for the Development of Scientific and Economic Excellence) on "Platform Independent Mobile Augmented Reality" (PIMAR).

In the state-of-the-art process of using these meta-tools, the developer analyzes one or several reference applications and extracts knowledge to specify the IDE components. This approach, referred to as *bottom-up development* [9], assumes that full reference applications are provided upfront, which is reasonable if the involved technologies and user requirements are stipulated at the beginning of the project. In rapidly evolving software domains, however, this assumption does not hold anymore: Due to changing user demands and underlying technologies, a DSML is exposed to evolution during its whole lifespan. The following research question arises: **How can domain-specific IDEs be developed systematically in the presence of modeling language evolution?**

In this paper, we aim to address this question. Our main contribution is an *agile bottom-up process* for the development of domain-specific IDEs, focussing on the co-evolution of a DSML, its editors and code generators. The key idea is to organize language evolution into fine-grained evolution steps: In each step, prototype models are employed to generate one or several application prototypes. The developer manually modifies the prototypes as required for the evolution step. Then, the IDE developer identifies aspects concerning the DSML, editors, and code generators. These aspects are used as input for their synchronous evolution. Afterward, the application prototype is no longer required. The process is not designed for, but can be aligned with a specific agile methodology, such as Scrum. As our second contribution, we integrate this process in the overarching vision of a *three-tier process model*, involving the development of meta-tools, tools and applications. Our third contribution is an *experience report* concerning the development of a DSML and domain-specific IDE for mobile applications.

The remainder of this paper is structured as follows: In Sect. 2, we present the process, outlining its main activities. Sect. 4 introduces the three-tier process model. Sect. 3 reports on experiences we made applying the process in a research project on the development of mobile applications. In Sect. 5, we discuss related work. In Sect. 6, we conclude and elaborate our plans for future work.

2 Agile bottom-up development of domain-specific IDEs

Domain-specific IDEs are an enabling technology for the model-driven development of specific applications. At a minimum, a domain-specific IDE comprises one or several model editors for the underlying DSML and code generators for one or several target platforms. Additional components may include version management, testing and debugging tools.

In this section, we give an overview of the proposed agile bottom-up IDE development process: First, to define an initial DSML and IDE, a *domain analysis* is carried out, involving the extraction of domain concepts and generator templates from existing reference applications. Second, in the course of *continuous language and IDE development*, developers perform evolution steps, including the generation and modification of prototypes and successive evolution of the DSML and IDE. Third, evolution steps may require a follow-up *migration* step to reconcile inconsistencies introduced in existing prototype app models during the evolution step. These app models are model-based descriptions of prototypes. For each of these activities, we outline the involved manual and automated tasks and the tools supporting these tasks.

2.1 Domain analysis

Different rationales can motivate a change to model-driven development: First, in large software projects, a lot of boilerplate code may exist due to use cases showing certain similarities. Second, a number of separate applications might show similarities in structure and behavior. Third, it may be required to deploy one individual application to several target platforms. In each of these scenarios, the abstraction level of development can be lifted by using DSMLs with code generation facilities. The initial step to establish such a DSML based on one or several reference applications is called *domain analysis*.

Domain analysis involves three steps: *Quality assurance*, *domain concept identification* and *template extraction*. *Quality assurance* ensures that the existing applications exhibit high quality, rendering them suitable as reference applications for code generation. This task involves the identification of anti-patterns and refactoring towards design patterns. During *domain concept identification*, concepts recurring throughout the reference applications are identified; they are reflected as model elements in the DSML. The aim of *template extraction* is to specify generator templates: A generator template represents a unit of code with gaps. The gaps are filled during application development by the generator, using application-specific information given by instances of the DSML.

Quality assurance can be partly automatized using static analysis tools supporting the detection of anti-patterns and code smells [10]. A promising technology to detect recurring concepts is automated clone detection [11]. To our knowledge, no specific tools exist to manage the extraction of templates based on reference applications, leaving it a fully manual step.

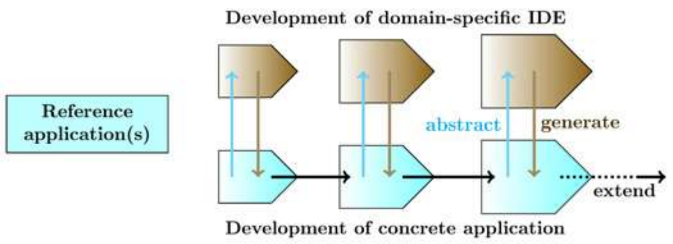

Development of domain-specific IDE
Reference application(s)
abstract
generate
extend
Development of concrete application

Fig. 1. Agile MDD process in action: Fine-grained evolution steps

2.2 Continuous language and IDE development

We propose to develop IDE components, notably the model editors and a code generators, in fine-grained iterations (cf. Figure 1): First, developers decide on the next feature that should be supported by the DSML and its IDE. Then, the IDE developer generates one or more prototypes from app models and manually extends these prototypes by the code required to implement this feature. The extension is then analyzed and results in a synchronous evolution step of the DSML and its IDE.

In this approach, the IDE developer is required to inspect the generated prototypes and extend them to incorporate new features. Therefore, it is essential that generated applications are working software systems and that the generated code is of good quality, i.e. well structured and easy to understand. As an aid to support the comprehension of the generated code, we provide a mapping between DSML elements and the individual code generator templates involved in implementing these elements. In our experience, such a mapping has proven itself valuable.

Various meta-tools allow specifying editors, transformations and further tools. GMF, Xtext and Sirius support high-level specification of graphical and textual model editors. ATL [12], Henshin [13], ViaTra [14] and many more support the specification of model translations, simulations and optimizations. There are further meta-tools for IDE components such as EMF Refactor [15] for model quality assurance tools, and EMFCompare [16] and SiLift [17] to support version management features. Since continuous language evolution results in continuous IDE evolution, co-evolution processes are important to be considered and to be supported by tools. Therefore, meta-tools are needed for migrating all dependent artifacts such as instance models, model transformations, especially code generators, model editor specifications, model quality assurance and version management tools. Future research is needed to automate these migrations.

2.3 Migration of app models

Since app models are directly dependent on the evolution of their DSML, they have to be kept consistent with the DSML. One possibility is to only make changes that do not necessitate adapting the software systems on lower layers. However, this might lead to compromise solutions in language design. The alternative is to migrate them accordingly which allows to freely develop DSMLs.

Co-evolution tools such as Edapt [18] and Flock [19] are available, but still show specific limitations: For instance, Edapt supports the evolution of meta-models using pre-defined operations and the automatic deduction of a suitable migration script for all instance models. However, integrating these pre-defined operations requires a significant adoption of existing modeling workflows and tools. As a consequence, migration processes are currently performed by hand, which can be tedious and error-prone. In the future, we aim to provide tool support for the automated co-evolution of app models. We intend to base these tools on results concering the co-evolution of language meta-models and instance models [20,21].

3 Experience Report

In this section we provide an experience report concerning the development of a DSML and a domain-specific IDE for mobile applications. First, we provide a domain description, including the resulting modeling approach. Second, to demonstrate the application of meta-tools within the proposed process, we describe one iteration of language and IDE development. Third, we report on continuous language and generator extension. To determine the usefulness of the application of meta-tools, we investigated how these tools can shorten iterations by allowing the automatic (re)generation of code for certain IDE artifacts. We tracked the size of an editor and a code generator and the number of covered use cases during the development of the IDE. Finally, we discuss the open problems.

3.1 Domain description

Along with our industrial partner *advenco* [22], a medium-sized software consulting company, we discovered the domain by analyzing two of their products. The first product is a multimedia guide for tourists, guiding them through places of interest such as museums, exhibitions or towns. A second product allows defining business processes using mobile and further devices. As a result, we define several characteristics of the application domain that should be supported by the IDE:

Core functionality:
- provide language elements for data, behavior, and GUI modeling
- generate apps that operate in multiple contexts (e.g., online or offline) and support user-provided content (e.g. current information about tourist events)

Enhanced functionality:
- provide access to device sensors (e.g. optical barcode and RFID tags)
- provide functionalities for augmented reality applications
- provide functionalities for e-Learning applications (e.g., vocabulary training, self-assesment, safety training, etc.)

Supporting new technology:
- new platforms and versions
- new kinds of devices (e.g., wearables, tablets, embedded Android, etc.)

On the basis of our domain analysis, we support different modeling aspects and generate native apps that can be flexibly configured by users. Figure 2 illustrates the resulting modeling approach. It comprises two code generators, one for Android and one for iOS, which generate runnable applications (100% of the application code is generated from the models). Detailed information regarding the abstract and concrete syntax of data, behavior, and GUI models is provided in [23].

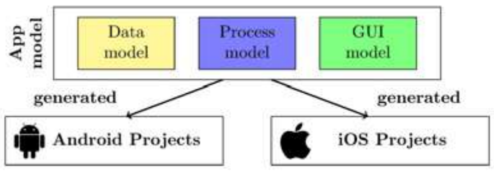
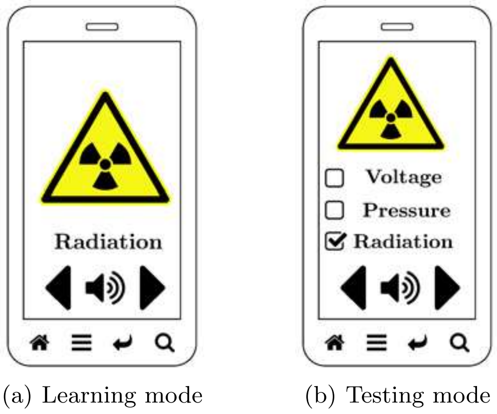


Fig. 2. MDD approach for mobile applications

3.2 Example: Language design and development iteration

To illustrate an evolution step, we implement an e-Learning application for safety instructions. The e-Learning application, illustrated in Figure 3, comprises two use cases: The first use case, called *learning mode*, concerns learning using different media types (e.g., videos, pictures and sound recordings). The second use case, called *testing mode*, allows to practice learned content using assignment tasks.



(a) Learning mode (b) Testing mode

Fig. 3. e-Learning application for a safety instruction

The GUI meta-model includes style settings and generic page types serving different purposes. For example, there is a *ViewPage* for displaying objects and an *EditPage* for modifying them. To offer the e-Learning functionality, we introduced an *ELearningPage* into the GUI meta-model. As shown in Figure 3, the purpose of the *ELearningPage* is to present learning content (*learning mode*) or provide a self-test format (*testing mode*). The *ELearningPage* hides the technical details (e.g., playing the sound file, loading the media files, etc.) from the modeler. Adding *ELearningPage* was the only DSML extension.

After having extended the modeling language, the visual editor was regenerated. Then, the code generators had to be adapted to the new language elements. In order to process the new *ELearningPage* element, a new template (*ELearningPageGenerator*) was added to the generator. This template initially generated an empty Android activity or iOS view. We then extended the empty mock class with the required code. After testing, we abstracted the inserted code to code templates. The iteration ended when the generated application fulfilled the same requirements as the extended prototype. The prototype is no longer required.

3.3 Continuous language and generator extensions

The entire process of IDE development usually contains several iterations of the above-mentioned kind. We have changed the DSML 26 times within a period of approx. 18 months to cover the 19 use cases of the two reference applications.

Most of the changes (19 times) were pure extensions of the DSML (from 25 up to 46 elements). We have developed five case studies on mobile apps of different kinds simultaneously through the course of the development of the IDE.

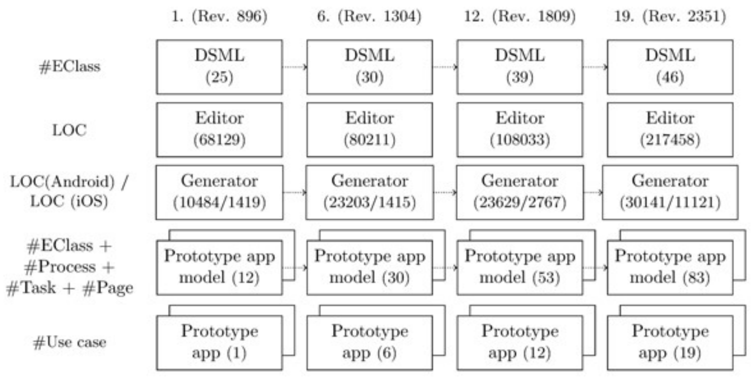


Fig. 4. Continuous language, IDE and prototype extensions

Figure 4 shows four samples of the 19 iterations; we can see the incremental growth of both the IDE and the prototype (generated from the app model). Each iteration realizes an additional use case. During the first six iterations (Rev. 1304), the core functionalities were implemented, followed by the enhanced functionalities.

After each DSML modification, the visual model editor, a key component of the domain-specific IDE, was (re)generated. Figure 4 shows the growth in size of the visual model editor and the Android code generator: In terms of LoC, the model editor was at least three times larger than the code generator during all iterations. Being able to generate this large percentage of the codebase helped to shorten the development cycles: Changes to the DSML were immediately available in the editor. The effect of these changes to the visual editor allowed early detection of language design flaws and appropriate refactorings.

Besides, both platform-specific code generators were extended manually. The iterative approach reduced the complexity of generator construction and extension since developers could focus on recent changes made to the DSML. There were exceptions in the form of cross-cutting modeling elements (e.g., application-wide style settings), which affected many generator templates.

Another advantage of our approach was the co-evolution of the IDE and the app model. IDE users could create app models at an early stage of IDE development and provide test models for the code generators. Figure 5 shows the Eclipse-based domain-specific IDE (visual editor) resulting from continuous language and generator extensions.

3.4 Threats to validity and open problems

Considering *external validity*, it is not ensured that our experiences generalize to arbitrary syntactic and semantic changes in DSMLs. To mitigate this threat,

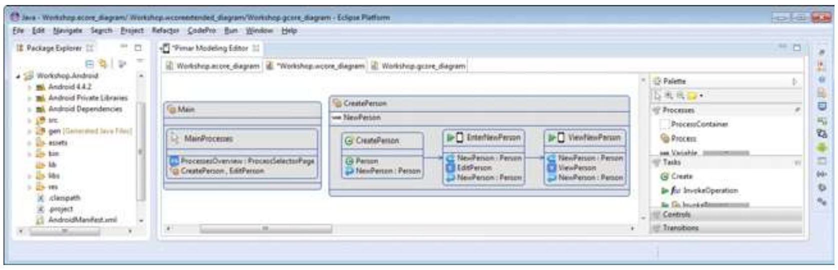


Fig. 5. Domain-specific IDE for model-driven development of mobile applications.

we consider a wide range of use-cases and code generation for multiple target platforms. More experiments are needed to allow general conclusions about the utility of the approach. A threat to *internal validity* is the lack of a case study using traditional processes. Still, we argue that these processes do not provide a course of action to support the co-evolution of DSMLs and IDEs, which renders them unsuitable for our scenario.

The automated deduction of migration scripts for all dependent artifacts is an unsolved problem: For example, changes to the DSML affect the respective templates and existing app models. Currently, all modeling artifacts are migrated manually after each iteration step, a time-consuming and error-prone task.

4 Three-tier agile process model

From a global perspective, taking into account all processes and tools involved in model-driven development leads to three tiers of software development: The development of concrete applications, the development of domain-specific IDEs for model-driven development, and the development of meta-tools to specify IDE components. Although the main contribution of this paper is an agile development process for domain-specific IDEs, we argue that concrete applications and meta-tools shall be developed based on agile principles as well. To quickly respond to new user demands and technologies, all involved software systems should be developed continuously. Their development should incorporate short feedback cycles based on running software.

This set of requirements leads to the stipulation of a three-tier agile development process model, outlined in Figure 6. In the domain of mobile applications, for example, a concrete application is a mobile app that is developed using an IDE for model-driven development of mobile apps. Meta-tools such as editor generators or model-to-text transformation approaches can be used to specify model editors and code generators of these IDEs. The interplay of three different kinds of software projects leads to challenges: Changes in one software project can affect the other ones. These challenges are aggravated by different life cycles and change frequencies. While applications are quickly developed by model-driven development, IDE development is much slower, and meta-tools are usually developed completely independently of concrete IDEs.

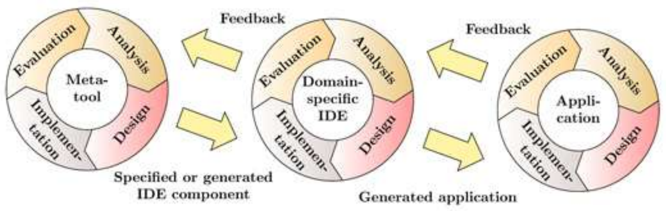


Fig. 6. Three-tier agile software development process model

5 Related work

Several papers describing agile model-driven development, such as [3,4], focus on developing application software by using existing IDEs for MDD, whereas our scope extends to agile development of the domain-specific IDEs themselves.

Völter [24] reflects on best practices for model-driven engineering and DSMLs. Our approach incorporates many of these practices and joins them to the agile development process of domain-specific IDEs. We also focus on minimizing manually written code and co-evolving our language and concepts.

Bagheri et al. [9] propose extracting partial models from stable parts of the system, leaving the remaining parts to be developed manually. They support combining generated and manually written code through their technique called *partial synthesis*. In our process, at the end of an iteration, all manually written code is eventually integrated in the generator, app model, or DSML. Thus, the resulting prototype only contains code produced by the generator.

Fehrenbach et al. [25] consider software evolution by introducing an embedded DSL into a legacy code base. Their approach is orthogonal to ours which assumes that the DSL itself is subject to evolution.

6 Conclusion

We propose an agile bottom-up development process for domain-specific IDEs supporting model-driven development. Starting from reference applications, the domain is analyzed. Prototype applications are continuously generated and extended, leading to continuously evolving domain-specific modeling languages and supporting IDEs. The greater vision is a *process model* for the integrated agile development of applications, domain-specific IDEs and meta-tools. Continuous evolution of all involved software artifacts is key to realize this vision.

As future work, we aim to integrate isolated IDEs for different domains. In an upcoming case study, we will examine the integrated development of data-centric and gaming applications sharing a set of communication points in the IDE and application layers. To specify the communication points, we extended the EMF meta-tool infrastructure with an interface concept [26], allowing us to generate intents between activities and/or services of the involved apps.

References

1. Thomas Stahl and Markus Völter. *Model-Driven Software Development.* Wiley, 2006.
2. Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.
3. Yuefeng Zhang and Shailesh Patel. Agile model-driven development in practice. *IEEE Software*, 28(2):84–91, 2011.
4. Vinay Kulkarni, Souvik Barat, and Uday Ramteerthkar. Early experience with agile methodology in a model-driven approach. In *14th Int. Conf. on Model Driven Engineering Languages and Systems*, pages 578–590, 2011.
5. Graphical Modeling Framework. `http://www.eclipse.org/gmf`.
6. Eclipse Sirius. `http://www.eclipse.org/sirius`.
7. Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt Publishing Ltd., 2013.
8. EGL Development Tools (EDT). `http://www.eclipse.org/edt`.
9. Hamid Bagheri and Kevin J. Sullivan. Bottom-up model-driven development. In *35th Int. Conf. on Software Engineering*, pages 1221–1224. IEEE/ACM, 2013.
10. Jernej Novak, Andrej Krajnc, and Rok Zontar. Taxonomy of static code analysis tools. In *MIPRO, 2010 Proc. of the 33rd Int. Conv.*, pages 418–422. IEEE, 2010.
11. Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.
12. Eclipse ATL. `http://www.eclipse.org/atl`.
13. Eclipse Henshin. `http://www.eclipse.org/henshin`.
14. Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214–234, 2007.
15. EMF Refactor. `http://www.eclipse.org/emf-refactor`.
16. Eclipse EMF Compare. `http://www.eclipse.org/emf/compare`.
17. Timo Kehrer, Udo Kelter, and Gabriele Taentzer. Consistency-Preserving Edit Scripts in Model Versioning. In *28th IEEE/ACM Int. Conference on Automated Software Engineering*, pages 191–201. IEEE, 2013.
18. Eclipse Edapt. `http://www.eclipse.org/edapt`.
19. Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Model Migration with Epsilon Flock. In *3rd Int. Conf. on Theory and Practice of Model Transformations*, pages 184–198. Springer, 2010.
20. Boris Gruschko, Dimitrios Kolovos, and Richard Paige. Towards Synchronizing Models with Evolving Metamodels. In *Ws. on Model-Driv. Softw. Evolution*, 2007.
21. Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *12th Int. Conf. on Enterprise Distributed Object Computing*, pages 222–231. IEEE, 2008.
22. Advenco Consulting GmbH. `http://www.advenco.de`.
23. Steffen Vaupel, Gabriele Taentzer, Jan Peer Harries, Raphael Stroh, René Gerlach, and Michael Guckert. Model-driven development of mobile applications allowing role-driven variants. In *17th Int. Conf. on Model-Driven Engineering Languages and Systems*, pages 1–17, 2014.
24. Markus Völter. Md* best practices. *J. of Object Technology*, 8(6):79–102, 2009.
25. Stefan Fehrenbach, Sebastian Erdweg, and Klaus Ostermann. Software evolution to domain-specific languages. In *Software Language Engineering*, pages 96–116. Springer, 2013.
26. Daniel Strüber, Gabriele Taentzer, Stefan Jurack, and Tim Schäfer. Towards a distributed modeling process based on composite models. *Fundamental Approaches to Software Engineering*, pages 6–20, 2013.

Type Inference Using Concrete Syntax Properties in Flexible Model-Driven Engineering

Athanasios Zolotas[1], Nicholas Matragkas[2], Sam Devlin[1], Dimitrios S. Kolovos[1], and Richard F. Paige[1]

[1] Department of Computer Science, University of York, York, UK
[2] Department of Computer Science, University of Hull, Hull, UK
`Email: {amz502, sam.devlin, dimitris.kolovos, richard.paige}@york.ac.uk, n.matragkas@hull.ac.uk`

Abstract. In traditional Model-Driven Engineering (MDE) models are instantiated from metamodels. In contrast, in Flexible MDE, language engineers initially create example models of the envisioned metamodel. Due to the lack of a metamodel at the beginning, the example models may include errors like missing types, typos or the use of different types to express the same domain concepts. In previous work [1] an approach that uses semantic properties of the example models to infer the types of the elements that are left untyped was proposed. In this paper, we build on that approach by investigating how concrete syntax properties (like the shape or the color of the elements) of the example models can help in the direction of type inference. We evaluate the approach on an example model. The initial results suggest that on average 64% of the nodes are correctly identified.

1 Introduction

In traditional MDE the models instantiated using editors, conform to a predefined metamodel. In contrast, in Flexible MDE, language engineers use drawing editors, like those proposed in [2], [3] and [17], to express example models that will be used to infer the envisioned metamodel. These tools are used as free-form whiteboards on which the domain experts sketch elements that represent concepts and assign types to them. As there is no metamodel to specify the semantics, this process is error prone. Drawn elements may be left untyped, or the same concept may be represented by using two or more types. The first could happen either unintentionally (the engineer forgot to assign a type to the element) or intentionally (engineers leave repeatedly-expressed concepts untyped). The latter may occur either because two or more engineers are involved and thus different terminology may be used or, if the process is long-term, the engineer may have forgotten the type used in the past for a specific concept.

This paper addresses the challenges associated with identifying and managing omissions during type assignment in flexible modelling by using a variation of the *type inference* approach proposed in [1]: missing types are inferred by computing and analysing matches between untyped and typed elements that share the same *graphical* characteristics. The difference between the approach proposed in [1] is that in this work we are looking for concrete syntax properties of the drawings and not at semantic relations. More specifically, in this approach, features that represent graphical characteristics of the nodes (i.e. the shape, color, width and height) are fed to a Classification and Regression Tree (CART) algorithm which predicts the types of the untyped nodes. We present the approach in detail, using an illustrative flexible modelling approach based on GraphML and the flexible modelling technique called *Muddles* [2]. We demonstrate the approach's accuracy via experiments run on an example flexible model.

This approach is based on the assumption that language engineers tend to use, to the extent possible, the same concrete syntax to express the same concept in a diagram. However, in order to explore the capabilities of the approach in cases where this assumption does not stand (or stands partially) we add noise to the example model by applying changes to the graphical properties of its nodes (e.g. randomly changing the shape of some nodes).

2 Related Work

In [4], rules that should be taken into consideration to construct the graphical syntax of languages is proposed. The author claims that the importance of graphical notation is a neglected issue so far and he adapts the *theory of communication* by Shannon and Weaver [5] to highlight that the effectiveness of the graphical syntax can be increased by choosing the most appropriate notation conventions of these that the human mind can process. In [6], Bertin identified a set of 8 visual variables that can encode and reveal information about the elements they represent or their relations in a graphical way.

In the field of bottom-up metamodelling, Cho et al. [7] propose an approach for an semi-automatic metamodel inference using example models. In [8], example models created by domain experts can be used to infer the metamodel. In [9], evolved models that do not conform to their metamodel are used to recover the metamodel. In [2], users, using a drawing tool, define example models which are then amenable to programmatic model management scripts.

Type inference is used in programming languages. The Hindley-Milner [10] [11] algorithm and its extension by Milner and Damas [12] are the most common used in this domain. Code statements are reduced to simpler constructs for which a type is already known. Such approaches are challenging to apply in flexible modelling where there is no predefined abstract syntax. Inferring types (or metamodels) from example models is a *matching* problem: elements that are "sufficiently similar" may have similar or identical types; a classification was published in [13]. A model matching technique is used in [14] to generate traceability links. The nodes of the two models are matched by checking their *name similarity.* In [15] each element's unique MOF identifier is used to calculate the difference and union of models that belong to the same ancestor model. In [16] manually generated signatures are used to match elements and perform model merging. The last three approaches are of limited flexibility as they depend on names or persistent identifiers for inference. Finally, in Flexisketch [17] the authors adapt the algorithm proposed in [18] to find possible matches between the shapes of hand-drawn elements that appear on the same canvas.

This work builds on the approach presented in [1]. A CART algorithm is used to infer the types of nodes. However, the data fed to the classification algorithm consist of features that are related to semantic aspects of the example models (number of attributes, unique incoming and outgoing references and unique parents and children of each node). In this work, we propose the use of features that are related to the concrete syntax of the drawn example models.

3 Background: Muddles

In Muddles [2], a GraphML compliant drawing editor called yEd[3] is used to allow language engineers draw example models that are amenable to model management suites. The drawn elements are annotated with their types. Attributes can also be added by using the appropriate node properties input boxes. The drawing is automatically transformed to the intermediate Muddle model and can then be consumed by a model management suite like the Epsilon platform [19].

In this work, due to the fact that we are interested in the graphical properties of the drawings, like the shape, the color and the size of the drawn elements we use the extended version of the Muddles approach that was presented in [20] that extracts such information. A *muddling* example follows for better understanding.

3.1 Example

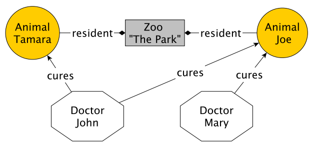


Fig. 1. Example

The language engineer is interested in creating a language for expressing zoos. The process starts by drawing an example zoo diagram (Fig. 1(b)). Next, diagram elements are annotated with basic type information. For instance, the type of both hexagon shapes is defined as *Doctor* and the type of the directed edges from *Doctor* to *Animal* nodes (circles) as instances of the *cures* relationship. The types are not bound to the shape but to each element, meaning that in another example or even in the same drawing, a hexagon can be of type *Doctor* while a different hexagon can be of type *Animal.* Types and properties of the types (e.g. attributes, multiplicity of edges) can also be specified. More details about these properties are presented in [2].

Model management programs can then be used to manage this diagram. For example, the following Epsilon Object Language (EOL) [21] script returns the

[3] `http://www.yworks.com/en/products_yed_about.html`

names of all the elements of Type *Animal* (the nodes typed as “Animal” have a String attribute named *name* assigned to them).

```
var animals = Animal.all();
for (a in animals) {
  ("Animal: " + a.name).println();
}
```

Listing 1.1. EOL commands executed on the drawing

4 Type Inference Approach

In this section the type inference approach followed is presented (an overview is presented in Fig. 2). The engineer creates an example diagram of the envisioned DSL using a GraphML compliant tool (yEd). Some of the elements may be left untyped for the reasons discussed in Section 1. The mechanism proposed in this approach, analyses the drawing and collects graphical information from it which is then fed into a classification algorithm to classify the untyped elements.

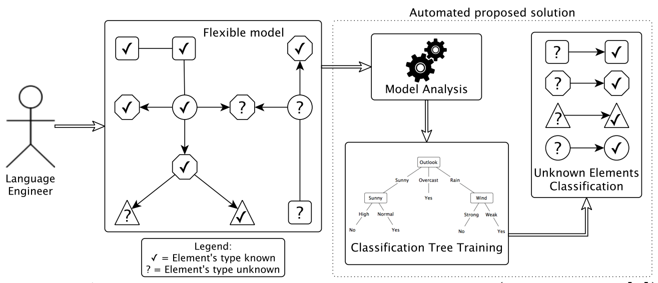

Automated proposed solution
Flexible model
Model Analysis
Language Engineer
Classification Tree Training
Unknown Elements Classification
Legend:
✓ = Element's type known
? = Element's type unknown

Fig. 2. An overview of the proposed approach (taken from [1])

4.1 Features Collection

The classification of the elements is based on characteristics that each element has. These are called *features*. We call the set of all the characteristics that are collected from each node as *feature signature*. In addition, the type of each node is attached to the last position of the feature signature. If the element is left untyped, then this place is left empty. In this approach we propose the use of 4 graphical characteristics as features for each node. The selected characteristics are presented in Table 1. Example signatures follow for better understanding.

Table 1. Signature features for nodes

Name of Feature	Description
Shape	The shape of the node (e.g. rectangle, ellipse, etc.).
Color	The color of the filling of the node in HEX (e.g. #FFCC00, etc.).
Width	The width of node expressed in pixels.
Height	The height of the node expressed in pixels.

For example, the feature signature for the node named “Animal Tamara” and annotated with the type “Animal” is [ellipse, #FFCC00, 114, 112, Animal]. The first value is the shape of the node (ellipse) while the second is the color of the filling. The third and fourth features are the width and the height. The last value is the annotated type for this element. Similarly, the feature signature for the node named “Animal Joe” is [ellipse, #FFCC00, 107, 105, Animal]. From these two signatures one can see that elements of the same type may or may not have the same signatures. This justifies the selection of a classification algorithm and not a simple matching algorithm, as classification algorithms do not only look for perfect matches but can also classify the items by picking each time the features that are more important in the set that they are trained on.

A script that parses the diagram and constructs the feature signature for the nodes was created. The signatures are stored in a file that is fed to the CART.

4.2 Classification

Classification algorithms are a supervised machine learning method for finding functions mapping input features to a discrete output class from a finite set of possible values. They require a dataset to train on, after which they can generalise from the previous examples given in the training set to predict the class of new unseen instances.

Many classification algorithms exist, some of the most established being decision trees, random forests, support vector machines and neural networks [22]. For this work we chose to use decision trees. Specifically, we used the rpart package (version 4.1-9)[4] that implements the functionality of (CART) [23] in R[5]. In practice other classification algorithms (i.e. support vector machines or neural networks) can often have higher accuracy, but will produce a hypothesis in a form that is not as easily interpreted. Given the exploratory nature of this work, these other algorithms were not deployed in favour of the aid to debugging provided by being able to interpret how the learnt hypothesis would classify future instances. For example, a possible decision tree for type inference is illustrated in Figure 3. Internal nodes represent features (e.g. Shape, Colour, etc.), each branch from a node is labelled with values of the feature and leaf nodes represent the final classification given. To classify a new instance, start at the root node of the tree and take the branch that represents the value of that feature in the new instance. Continue to process each internal node reached in the same manner until a leaf node is reached. The predicted classification of the new instance is the value of that leaf node. For example, given the tree in Fig. 3, a new instance which shape is not an ellipse and its colour is different than white (#FFFFFF) classified as Zoo (path is highlighted in Fig. 3).

In our approach, the feature signatures list that contains the signatures of the known elements of the model are the input features to the CART algorithm. A trained decision tree is produced which can be used to classify (identify the type of) the untyped nodes using their feature signatures. The success of a classification algorithm can be evaluated by the accuracy of the resultant model (e.g. the decision tree learnt by CART) on test data not used when training. The accuracy of a model is the sum of true positives and negatives (i.e. all correctly classified instances) divided by the total number of instances in the

[4] http://cran.r-project.org/web/packages/rpart/index.html
[5] http://www.r-project.org/

test set. A single measure of accuracy can be artificially inflated due to the learnt model overfitting bias in the dataset used for training. To overcome this, k-fold classification can be implemented [24]. This approach generates k different splits of the data into training and test data sets and returns the mean accuracy generated from k repeats with each repeat using a unique split of the data.

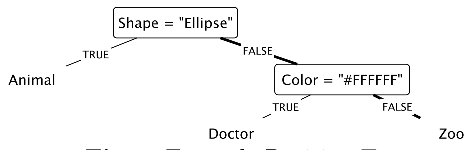


Fig. 3. Example Decision Tree

5 Experimentation Process

In this section the experimentation process used to evaluate the performance of the proposed approach is presented. An overview is given in Fig. 4[6].

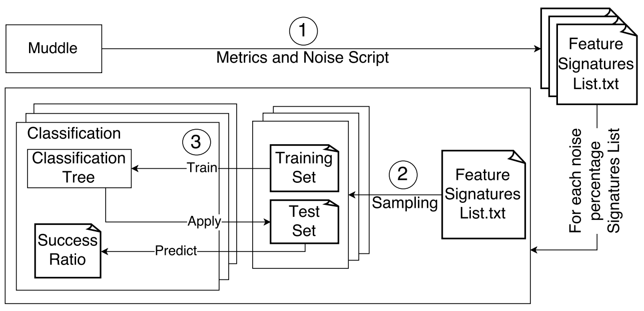


Fig. 4. The experimentation process

In order to test the proposed approach we applied the classification algorithm to a muddle. This muddle was created before commencing this research as part of a side project to express requirements for a web application. Our experience working with Muddles suggests that it is a fairly complicated example as it consists of more than 100 elements of 20 different types. This muddle was also one of the 11 models that were part of the experiment for the approach presented in [1]. A comparison with the 10 other models used in the experiment of the previous approach is not possible as these were automatically generated using mechanisms that are biased in the selection of the 4 features that we assess in this work: all the nodes were of the same shape, color and size.

In addition, we tested the resilience of the proposed approach to human error and the bias that our muddling habits may cause: we tend to use the same shape when we express a specific type. We need to highlight here, that in some cases all elements of the same type share same features but there are types where this is not the case. In contrast, some feature values (e.g. rectangle) were not used only for one type: elements of different types share common features in our

[6] The code, data, results and a guide on how to use the approach can be found in `http://www.zolotas.net/type-inference-graphical`

experimentation example. Arguably, this is the case with any other relatively large muddle as the number of available shapes and colors is in practice limited. Thus, in order to check if adhering to some basic conventions when drawing an example model is important for the accuracy of the prediction, we performed a second experiment by adding noise to some of the elements by explicitly changing their features. We did that gradually by altering randomly *one* feature of none (0%) up to all (100%) of the elements of the muddle using a step of 20% (0%, 20%, ..., 100%). 40% noise addition means that 40% of the nodes have *exactly one* feature (randomly selected) changed to something else (e.g. shape is changed from rectangle to ellipse). A step-by-step description of the experiment follows.

Initially a script collects the features from the muddle and places them in a list that includes the feature signatures for each element (step ①). As the example has 105 nodes, there are 105 feature signatures in this list. This process is repeated 6 times; a new signatures list file is created for each of the noise levels. Next, each list is randomly separated into a training and a test set (step ② in Figure 4). The training set contains the nodes for which in a realistic scenario the types are known while the test set contains those nodes that are left untyped. In order to reach unbiased results (due to an unlucky or lucky random sampling) we perform the random sampling 10 times for each file (10-Fold). It is also of interest to identify if the amount of knowledge that the algorithm has on each diagram is of importance to the success ratio. For that reason we use 7 different sampling rates; from 30% to 90%. For example, a 40% sampling rate means that 40 % of the nodes are thought to be of known type while the rest (60%) are the nodes for which the type is unknown. The generated couples of training and test sets (420 in total) are then fed to the CART algorithm (step ③). The algorithm is trained on a training set and predicts the types of the elements of the coupling test set. The success ratio of the prediction is then calculated.

6 Results

Table 2 summarises the results for all the 420 runs. Each cell in the table contains the average accuracy of the classification for the 10 runs for the specific added-noise level and sampling rate. For instance, the highlighted value **0.64** indicates that on average, 64% of the missing types were successfully predicted for the 10 samples of the 20% added-noise model, using a 70% sampling rate.

The results suggest that the accuracy scores for the original (0% noise) model on average are similar to the success rates scores of the inference approach proposed in [1] (last row of Table 2). As expected, increasing the level of noise results in worse prediction scores. The same trend is also visible when decreasing the sampling rate. The correlation coefficients Corel. 1 and Corel.2, which definition follows, confirm that visual observation.

Corel. 1: How strong is the dependency between the sampling rate and the success score?

Corel. 2: How strong is the dependency between the added-noise level and the success score?

The values for Corel. 1 indicate a strong correlation for all the added-noise levels: prediction scores increase as training sets (the nodes with known types) become larger. The same behaviour was also observed in [1]. Regarding the second correlation (Corel. 2) we observe a perfect (negative) correlation between the number of nodes in a drawing that have altered features and the success score across all the sampling rates. This is evidence that following specific rules in the graphical syntax of the drawing increases the chances for correct type inference. By the term “specific rules” it is not implied that these rules should be strict. As discussed in previous sections, in the 0% added-noise example the authors use the same shapes to express the same concepts or in other cases the same color but not in a rigorous manner: same graphical properties are used in different concepts while the same concepts may have different graphical properties. We have also identified that the results related with the added noise experimentation (especially those of 80% and 100%) are influenced a lot by the randomness in picking the feature that each time will be altered. More specifically, if the random algorithm picked a lucky noise injection (changing in each node a feature that it is not distinctive for this type) then the results were better.

Table 2. Results summary table

	Average Success Ratio (Accuracy) for Different Sampling Rates								
Noise Level	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Corel. 1**
0%	0.55	0.58	0.61	0.63	0.67	0.71	0.71	0.637	0.99
20%	0.50	0.53	0.58	0.61	**0.64**	0.63	0.74	0.604	0.96
40%	0.47	0.53	0.52	0.56	0.57	0.62	0.63	0.557	0.97
60%	0.42	0.46	0.45	0.49	0.46	0.56	0.53	0.481	0.85
80%	0.35	0.38	0.44	0.43	0.50	0.44	0.56	0.443	0.89
100%	0.35	0.35	0.35	0.37	0.42	0.42	0.40	0.380	0.85
Corel. 2	-0.98	-0.98	-0.99	-0.99	-0.95	-0.98	-0.93	-	-
Muddle [1]	*0.55*	*0.56*	*0.59*	*0.65*	*0.59*	*0.67*	*0.64*	*0.607*	-

6.1 Threats to Validity

One example, created before commencing this research, was used to evaluate the approach. Although having one example may not be the best way to extract safe results we believe that it gives at least preliminary evidence that concrete syntax can be used to infer types of nodes in flexible modelling. A threat related to that which works against the approach is that the model consists of 20 different types. According to our experience with Muddles, this is a marginal one as engineers tend to use more rigorous editors as the models increase in size. The results in the experiments run in the previous approach [1] suggest that as the number of types increases the prediction accuracy decreases. In addition, the example model consisted of a number of highly repeated elements of the same type (e.g. nodes of type ”MenuItem”). If these elements can be correctly identified, the fact that they participate a lot in the example leads to better success scores. However, a balancing fact is that there were also 5 types which appear less than 2 times, reducing the chances of having them predicted correctly in case all of their instances end in the testing set (there will be no appearance in the training set so the algorithm doesn't know about the existence of this specific type).

7 Conclusions and Future Work

In this work we assessed whether the *concrete syntax* of flexible models can be used to infer the types of the elements that are left untyped using CART. Experiments suggest that on average 64% of the types were correctly identified. We also experimented with the intentional addition of noise in the diagram to check how this affects the prediction accuracy. A strong correlation between the percentage of altered nodes and the accuracy was identified providing evidence that this approach is more successful if it is used under the assumption that modellers tend to use, to the extent possible, the same graphical notation for elements of the same concept. We believe that this behaviour can be "unintentionally" replicated because of the "copy-paste" nature of muddling (e.g. create an animal node once and then copy & paste the node when you need it again). This way the same graphic notation is used for all the elements of the same type. It is important to highlight that if the type of the node was typed before the "copy-paste" event took place, which is not necessarily always the case, then the type is also transferred to the newly pasted nodes.

In the future, we plan to introduce and test additional features like font size and color, border size, orientation etc. In addition, in order to check if the prediction accuracy can be further increased, our intention is to combine the concrete syntax feature with the semantic related features proposed in [1]. Finally, we plan to run user studies to further evaluate the approach on more example models developed by language engineers.

Acknowledgments

This work was carried out in cooperation with Digital Lightspeed Solutions Ltd, and was part supported by the Engineering and Physical Sciences Research Council (EPSRC) through the Large Scale Complex IT Systems (LSCITS) initiative, and by the EU, through the MONDO FP7 STREP project (#611125).

References

1. Zolotas, A., Matragkas, N., Devlin, S., Kolovos, D., Paige, R.: Type inference in flexible model-driven engineering. In Taentzer, G., Bordeleau, F., eds.: Modelling Foundations and Applications. Volume 9153 of Lecture Notes in Computer Science. Springer International Publishing (2015) 75–91
2. Kolovos, D.S., Matragkas, N., Rodríguez, H.H., Paige, R.F.: Programmatic muddle management. XM 2013–Extreme Modeling Workshop (2013) 2
3. Gabrysiak, G., Giese, H., Lüders, A., Seibel, A.: How can metamodels be used flexibly. In: Proceedings of ICSE 2011 workshop on flexible modeling tools, Waikiki/Honolulu. Volume 22. (2011)

4. Moody, D.L.: The physics of notations: toward a scientific basis for constructing visual notations in software engineering. Software Engineering, IEEE Transactions on **35**(6) (2009) 756–779
5. Shannon, C.E., Weaver, W.: The mathematical theory of communication. (2002)
6. Bertin, J.: Semiology of graphics: diagrams, networks, maps. (1983)
7. Cho, H., Gray, J., Syriani, E.: Creating visual domain-specific modeling languages from end-user demonstration. In: Modeling in Software Engineering (MISE), 2012 ICSE Workshop on, IEEE (2012) 22–28
8. Sánchez-Cuadrado, J., De Lara, J., Guerra, E.: Bottom-up meta-modelling: An interactive approach. Springer (2012)
9. Javed, F., Mernik, M., Gray, J., Bryant, B.R.: Mars: A metamodel recovery system using grammar inference. Information and Software Technology **50**(9) (2008) 948–968
10. Hindley, R.: The principal type-scheme of an object in combinatory logic. Transactions of the american mathematical society (1969) 29–60
11. Milner, R.: A theory of type polymorphism in programming. Journal of computer and system sciences **17**(3) (1978) 348–375
12. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM (1982) 207–212
13. Kolovos, D.S., Di Ruscio, D., Pierantonio, A., Paige, R.F.: Different models for model matching: An analysis of approaches to support model differencing. In: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models. CVSM '09, Washington, DC, USA, IEEE Computer Society (2009) 1–6
14. Grammel, B., Kastenholz, S., Voigt, K.: Model matching for trace link generation in model-driven software development. Springer (2012)
15. Alanen, M., Porres, I.: Difference and union of models. Springer (2003)
16. Reddy, R., France, R., Ghosh, S., Fleurey, F., Baudry, B.: Model composition-a signature-based approach. In: Aspect Oriented Modeling (AOM) Workshop. (2005)
17. Wüest, D., Seyff, N., Glinz, M.: Flexisketch: A mobile sketching tool for software modeling. In: Mobile Computing, Applications, and Services. Springer (2013) 225–244
18. Coyette, A., Schimke, S., Vanderdonckt, J., Vielhauer, C.: Trainable sketch recognizer for graphical user interface design. In: Human-Computer Interaction–INTERACT 2007. Springer (2007) 124–135
19. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.: The design of a conceptual framework and technical infrastructure for model management language engineering. In: Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on, IEEE (2009) 162–171
20. Zolotas, A., Kolovos, D.S., Matragkas, N., Paige, R.F.: Assigning semantics to graphical concrete syntaxes. In: XM 2014–Extreme Modeling Workshop. 12
21. Kolovos, D.S., Paige, R.F., Polack, F.A.: The epsilon object language (eol). In: Model Driven Architecture–Foundations and Applications, Springer (2006) 128–142
22. Jiawei, H., Kamber, M.: Data mining: concepts and techniques. San Francisco, CA, itd: Morgan Kaufmann **5** (2001)
23. Breiman, L., Friedman, J., Stone, C.J., Olshen, R.A.: Classification and regression trees. CRC press (1984)
24. Mitchell, T.M.: Machine learning. 1997. Burr Ridge, IL: McGraw Hill **45** (1997)

Flexible Modelling for Requirements Engineering

Athanasios Zolotas[1], Nicholas Matragkas[2], Dimitrios S. Kolovos[1], and Richard F. Paige[1]

[1] Department of Computer Science, University of York, York, UK
[2] Department of Computer Science, University of Hull, Hull, UK
Email: {amz502, dimitris.kolovos, richard.paige}@york.ac.uk, n.matragkas@hull.ac.uk

Abstract. Many applications that are developed do not completely fulfil the requirements of their stakeholders. This can be a result of inadequate requirements elicitation and poorly defined requirements. Many solutions, including model-driven inspired ones, have been proposed to improve the elicitation of the requirements, though many of them are not yet widely used in practice as they require training of both the employees and the stakeholders. In this paper we propose the use of flexible modelling for eliciting and capturing the requirements of applications to facilitate the production of correct products that deliver on the *contract* defined between clients and developers. We argue that the use of flexible modelling can lower the entry barrier for use in the industry. The proposed method, called FlexRE, is applied to a scenario to demonstrate its capabilities and ways it can be extended.

1 Introduction

Tendered contracts [1] are very often used for the 'first contact' between the clients and the companies which develop applications. An initial, unstructured set of needs is given in the form of a tendered contract to candidate companies. Developers bid for the project by proposing an estimation on cost and time needed for the realisation of the project. The clients pick the solution that appears to be the most appropriate. After the agreement, business analysts, following different requirements elicitation techniques (e.g. interviews, prototypes, etc.) add more details to the requirements to let the developers have a better understanding on the needs of the clients. Many projects that follow this process fail to match the real client needs. Different studies [2], [3], [4] show that one of the reasons for that problem is the ambiguity of the requirements.

Many techniques have been developed to tackle this problem, including those that are model-driven inspired. The Agile methodologies, were proposed to increase the active participation of the client in the software development process, thus in the requirements elicitation phase. The above research surveys suggest that the problem still exists.

The proposed solutions so far urge the composition of either a very structured artefact which should conform to a rigorously-defined metamodel or a totally unstructured composition of requirements using text. The former are difficult to be used by non-technical stakeholders, as they force them to follow the semantics that are bound to a fixed metamodel. Thus there is low clients' contribution to the requirements specification document and high entry barrier of the methodologies in real world. In contrast, the latter offer no or limited structure, thus one cannot benefit from the use of model management techniques.

This paper presents a novel approach in expressing tendered contracts that promotes the active involvement of the clients in the composition of the document having as primary goal to lower the entry barrier and reduce the cost of using Model-Driven Engineering (MDE) in the requirements elicitation phase. This approach, offers a varying range of structure, positioning itself between the unstructured and rigidly structured requirements methodologies. It is based on the GraphML [5] standard and the flexible modelling technique introduced in [6]. Due to specific interest from our industrial partner in Web applications, we applied and tested the approach in this domain. There were no indications that it could not be applied to other domains, like desktop or mobile applications.

2 Background and Motivation

Clients are usually domain experts; they generally can understand the terminology and the processes that take place in the domain. By contrast, business analysts, modellers and developers are technology experts and know how systems are built. The lack of knowledge of the domain by technology experts and the lack of knowledge of the technology by domain experts is termed as the *symmetry of ignorance*. [7] "Contributory methods" like prototypes and scenarios, were added to the development processes to increase the active client's involvement and cooperation with the system's modellers. [7].

In WebML [8], probably the current standard in modelling web applications and their workflows, non-technical stakeholders should use elements from a palette that they are not familiar with as these elements were defined to represent technical concepts, like edges that represent successful and unsuccessful messages between actions or nodes that add/remove entries in ER Data diagrams.

A second problem of approaches that are based on rigorous and pre-defined metamodels is the fact that they restrict users of expressing requirements that the author of the metamodel did not think about. Thus, the scope of the projects that they can model is reduced (e.g. support for data-driven web applications only) or clients are restricted to use specific concepts that do not represent their envisioned applications. This boils down to the following anecdotal quote:

> "I've been in situations where I found that the modeling tool was simply too structured to let me describe everything I needed to describe."[3]

[3] http://programmers.stackexchange.com/questions/55679/why-arent-we-all-doing-model-driven-development-yet

A survey conducted among 12 Business Analysts working with IBM [9] verifies this argument and summarises the advantages and disadvantages of each of these two types of requirements elicitation approaches.

In this paper we argue for an approach that can increase the domain experts' contribution to the specification of the requirements. This approach is based on the use of a simple drawing tool and thus it requires less training given that one can build on familiar idioms and mechanisms reducing the cost of adaptation in the real world. It offers a varying range of structure, positioning itself between the unstructured and rigidly structured requirement methodologies promoting the use of Agile principles and processes.

3 Related Work

The first family of approaches includes those which use natural language as the means of expressing requirements. One approach is the use of spreadsheets or text documents for expressing the requirements. Controlled Natural Language (CNL) techniques aim to tackle the ambiguity of such statements written in natural language. In the same direction, Kaindl et al. [10] proposed the ReDSeeDS requirements language which uses dictionary definitions attached to the words and linking the words that refer to the same entity. These approaches can be used by non-technical stakeholders with no specific training increasing their contribution in the final definition of the system's requirements. However, they cannot benefit from the MDE advantages as they are not structured.

Two other approaches are the Business Process Modelling Language (BPMN)[4] and the Unified Modelling Language (UML)[5]. They allow business analysts to describe workflows of an application. The Web Requirements Engineering (WebRE) [11] is a UML profile specifically built to help with the specification of web applications. These diagrams are not usually used as the only way to describe the system but as supporting to the Software Requirements Specification, artefacts. The Navigational Developemnt Technique (NDT) [12] uses a template to store requirements. The Web Requirements Metamodel (WRM) [13] supports the definition of requirements using either UML Use Case Scenarios or NDT templates. All these approaches offer some structure, like links between the stakeholders and their needs, but not structure that is related with the elements that are used in fulfilling the requirements (e.g. images, buttons, menus, lists, etc.). In addition, training is needed to be used by non-technical stakeholders as they require conformance to specific concrete and abstract syntax.

The Program Design Language (PDL) [14] was the first attempt in the direction of using graphical interfaces to represent requirements, followed by other approaches, like the Structured Analysis and Design Technique (SADT) [15]. These techniques promote a very structured way in storing the requirements and can be seen as a design of the system under development rather than a SRS document thus can only be used by specialists. [16]

[4] http://www.omg.org/spec/BPMN/
[5] http://www.omg.org/spec/UML/

The closest to our approach is the one proposed in [9]. The authors, as mentioned in the previous section, surveyed 12 Business Analysts cooperating with IBM. As an outcome they proposed the use of a flexible modelling tools in the pre-requirements phase highlighting the advantages that such an approach has when domain experts and non-technical stakeholders are involved. In this paper, we propose the use of a flexible modelling approach not only in the pre-requirements but through the whole requirements phase that can also be used as a starting artefact for the other phases (e.g. coding, testing).

4 Flexible Modelling for RE

Our approach is based on the technical concepts of an approach called *Muddles*, proposed in [6], which employs flexible modelling techniques and promotes the use of a general-purpose drawing tool for the creation of programmatically manageable models. Flexible modelling allows the creation of models that are not instances of a specific metamodel. Models are created as an example to help the production of a rigorous metamodel that can then support MDE processes and model management using automated tools. Such a process allows domain experts to be actively involved in the creation of the metamodel by providing example models that describe their envisioned metamodel [6].

4.1 The Muddles Approach

The *Muddles* approach [6] uses a flexible graph definition language, GraphML [5], to allow language engineers to draw models and then annotate them so they can be accessed by model management suites. In the *Muddles* approach, each GraphML *Node* that is created in the drawing is extended with four Data fields attached to it (Fig. 1). The *Type*, is the field where the developer declares the type of each node. The *Properties*, is the field where the developer can add attributes to the node. For example, all the nodes of type "Web Page" have a String property called "Title". The *Default* field defines the variable which will be used by model management suites to access the label of the node. Finally, the *Contents* field defines the descriptor that will be used to get all the nodes contained within a parent node.

Each GraphML *Edge* is also extended by the Type, Properties and Default fields. In addition, fields to hold the descriptors of the "Source", "Target", the "Role in source", the "Role in target" and their multiplicities are defined. For example, an edge that represents linking between two nodes, has an outgoing relation named "linksTo" and an incoming relation called "linkedBy" with a multiplicity of 1. The "sourcePage" and "targetPage" define the keywords for the source and the taget page of the link, respectively. These keywords can be used by a model management suite to have access to the source or target element.

After the diagrams are annotated they are automatically transformed to an intermediate model which is instance of the *Muddle* metamodel. A discussion on the steps of the transformation are beyond the scope of this paper. The Epsilon

Data	
Type	Image
Properties	String src="..."
Default	name
Contents	children

(a) The Node Properties

Data			
Type	Link	Properties	String URL = ".."
Source	source	Role in sourcePage	linksTo1
Target	target	Role in targetPage	linkedBy1

(b) The Edge Properties

Fig. 1. The element properties

platform [17] they are using offers a driver that can consume *muddle* models and allow the execution of model management programs on them.

4.2 FlexRE

In this work we propose the use of a drawing tool that implements the GraphML standard to let domain experts in collaboration with business analysts, express the requirements of applications. By using a drawing tool and flexible modelling throughout the requirements engineering phase, we argue that the drawbacks (see Section 2) of current MDE techniques could be tackled. In addition, the drawings that include the requirements are structured and can be consumed by model management tools. We demonstrate this approach through a running example. In the example, we use the yEd[6] editor, the Epsilon platform [17] for model management and the technical concepts of the Muddles approach proposed in [6]. An overview of FlexRE is presented in Figure 2.

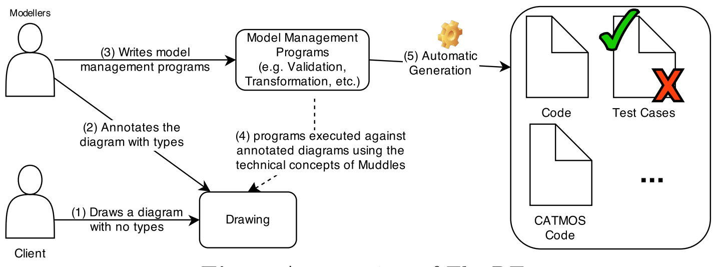

Modellers
(3) Writes model management programs
Model Management Programs (e.g. Validation, Transformation, etc.)
(5) Automatic Generation
Code
Test Cases
(2) Annotates the diagram with types
(4) programs executed against annotated diagrams using the technical concepts of Muddles
...
(1) Draws a diagram with no types
Drawing
CATMOS Code
Client

Fig. 2. An overview of FlexRE

In this case study, the client is interested in having a web application that will be used to present their Hotel company. It includes static and dynamic pages to present the rooms, the restaurant and help customers check for the rooms availability. A comprehensive set of the requirements are summarised in Table 1.

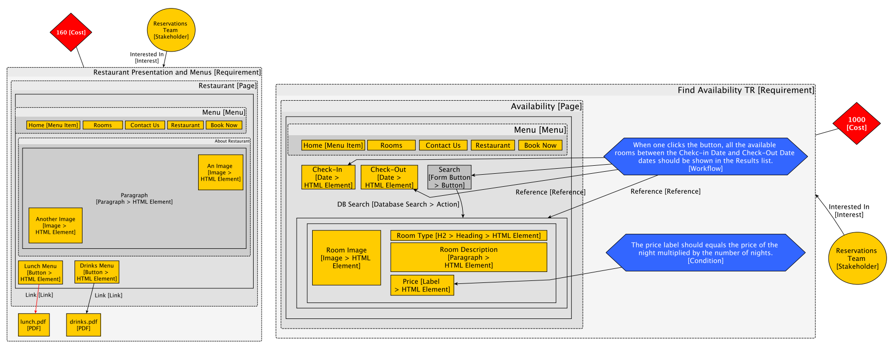
The *Manager*, who is the client in this scenario, starts drawing using the yEd editor, concepts that they are familiar with like check-in dates, check-out dates, etc. (step 1). He does not need to follow any specific rules and can use any shape or image from the palette of the drawing tool that represents their understanding of the required functionality. Business analysts can in parallel annotate the drawings with types that can be useful for the requirement methodologies they follow, using the fields presented in Fig. 1 (step 2). As an example, assume

[6] `http://www.yworks.com/en/products_yed_about.html`

Table 1. List of the requirements

ID	Stakeholder(s)	Description
FR1	Manager	"A page to present the rooms of the Hotel"
FR2	Manager	"A home page that will have a short introduction of the hotel, the rooms and the staff."
FR3	Manager, Customer Services	"A contact us page."
FR4	Manager, Reservations Team	"A page that will present all the available rooms for a specific range of dates."
FR5	Reservations Team	"A page for presenting the restaurant."

that, for the "FR5" requirement, the client ends up with the drawing shown in Fig. 3(a). The business analyst starts annotating these concepts. In the interest of a more clear presentation, the business analyst annotations for each element are shown in square brackets. In reality, the types are given using the Properties window of each element (Fig. 1). In this example, all the distinct HTML elements that the client uses are annotated as subclasses of the "HTML Element" class that this company uses as to identify them in the development process. This is done by using the ">" symbol which is used to denote the extension relationship. For instance, the Type property of the drawing that the client referred to as an image is set to "Image > HTML Element". These names given to the types are conventions, used to let the elements be accessed by a model management suite. It could be anything that fits the model-driven processes used in each company.


160 [Cost]
Reservations Team [Stakeholder]
Interested In [Interest]
Restaurant Presentation and Menus [Requirement]
Restaurant [Page]
Menu [Menu]
Home [Menu Item]
Rooms
Contact Us
Restaurant
Book Now
About Restaurant
An Image [Image > HTML Element]
Paragraph [Paragraph > HTML Element]
Another Image [Image > HTML Element]
Lunch Menu [Button > HTML Element]
Drinks Menu [Button > HTML Element]
Link [Link]
Link [Link]
lunch.pdf [PDF]
drinks.pdf [PDF]
Find Availability TR [Requirement]
Availability [Page]
Menu [Menu]
Home [Menu Item]
Rooms
Contact Us
Restaurant
Book Now
Check-In [Date > HTML Element]
Check-Out [Date > HTML Element]
Search [Form Button > Button]
DB Search [Database Search > Action]
Reference [Reference]
Reference [Reference]
When one clicks the button, all the available rooms between the Chekc-in Date and Check-Out Date dates should be shown in the Results list. [Workflow]
1000 [Cost]
Interested In [Interest]
Reservations Team [Stakeholder]
Room Image [Image > HTML Element]
Room Type [H2 > Heading > HTML Element]
Room Description [Paragraph > HTML Element]
Price [Label > HTML Element]
The price label should equals the price of the night multiplied by the number of nights. [Condition]

(a) The FR5 requirement drawing (b) The FR4 requirement drawing

Fig. 3. Two requirements

Non-functional requirements, business behaviour and workflows that can't be described using drawings can be typed in natural language. The business analysts can then either translate this into a drawing based on their experience or can simply attach bits of these requirements written in natural language to parts of the drawing that it is related to. For example, assume that the client needs to express the following requirement: "When the *button* is clicked, all the available rooms between the *check-in* and *check-out* dates should be shown in the *results list.*". This is part of the FR4 requirement shown in Fig. 3(b). The workflow that cannot be drawn is expressed using plain text inside the hexagons (see Fig. 3(b)) and can then be linked with the button that the client talks about, the check-in/out input fields and the results list that the client refers to.

The client continues drawing his understanding of the desired web application. As soon as the client has finished developers can then prepare model management programs or re-use those they have written in the past (step 3). These programs can be executed against the annotated drawings (step 4) using the technical facilities of the *Muddles* approach, to produce different artefacts that are interested in like pieces of code, test cases, textual contracts, and many others (step 5). Such examples are presented in the Section 5.

The same process could, in principle, be supported with a more traditional editor based on a fixed metamodel (e.g., derived using EMF/GMF). An advantage of using a flexible model is that it doesn't restrict clients in expressing concepts that the developers hadn't considered about when creating the metamodel. In other words, it allows clients to directly formulate the language of discourse without being restricted by a predefined set of constructs produced by non-experts. In addition, the drawing editor is a tool that even non-technical stakeholder arguably are able to use with no extensive training.

A first disadvantage of using flexible modelling is that it is error-prone. For instance, typos in the definition of the type of an element will create a new type. Secondly, there is a possibility of leaving untyped nodes which will left out from the model management processes. The above risks can be tackled by using validation rules. For instance, one could write a re-usable post-processor that calculates distance/similarity metrics for the Types. If the value for two Types is above a specific threshold it alerts the modellers for possible errors. Work in this direction is being carried out.

5 Application Scenarios

As discussed in the previous section, the annotated drawing can be consumed by programs expressed in model management languages of the Epsilon platform [6]. In this section we present some examples on how the drawings can be used to support different phases and aspects of the software development lifecycle.

5.1 Validation

It would be of interest to check if important rules/constraints of the domain are being met. For example, in the web domain, all the "Link" edges must have exactly one source and one target element. Conformance to such rules can be achieved by applying model validation to the drawing. In our scenario we use the Epsilon Validation Language (EVL), to validate that the above constraint (see Listing 1.1). The "source" and "target" keywords used in the code are the names the modellers provided in the *Source* and *Target* elements of the "Link" properties (see Fig. 1). Similarly, the keyword "linksTo" is the identifier of the *"Role in source"* property (see Fig. 1). The EVL statements, can be run against the drawings, are written once and can be re-used for all the projects.

```
context Link {
  constraint SourceAndTargetExist {
    guard : self.isTypeOf(Link)
    check : (self.linksTo.sourcePage.all.size == 1) and (self.linksTo.targetPage.all.
         size >= 1)
    message :  "Link edges should have exactly one source node and at least one
         target node"
  }
}
```

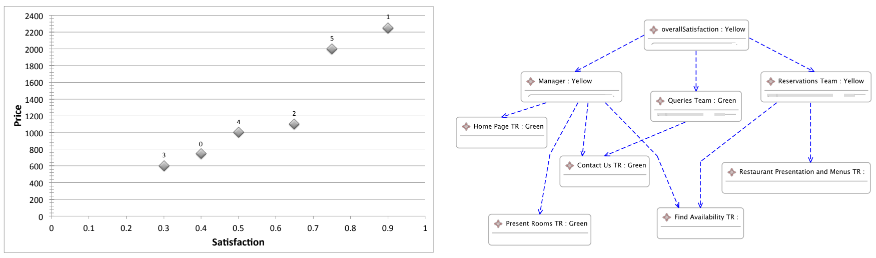
Listing 1.1. Example of EVL rule for validating the Link edges.

5.2 Next Release Problem

The Next Release Problem (NRP) is a multi-objective optimisation problem. The best set of requirements that should be implemented into the next software release, staying within the budget, is calculated. Burton et al. [18] offer a tool, as part of the Capability Acquisition Technique with Multi-Objective Search (CATMOS) methodology, that uses genetic algorithms and Pareto fronts to identify the optimal solutions calculating the possible trade-offs between the requirements of different stakeholders. The optimal solutions are visualised based on the total cost and total satisfaction they offer. Details on how the CATMOS suite works to find the optimal solutions are beyond the scope of this paper.

In this section we demonstrate the automatic transformation of the stored requirements into code that can be consumed by the CATMOS capability acquisition suite in [18] to solve the Next Release Problem.

Based on the scenario presented above, the developers just need to add and annotate two new types on the drawing. The first one is of type "Cost" (see Fig. 3) and defines the estimated cost to implement each component. The second is a "Dependency" edge and shows the dependency between two or more requirements. The drawing is now ready to be consumed by a model-to-text transformation expressed in the Epsilon Generation Language (EGL) and automatically generate the code needed to run the CATMOS tool. The EGL script is written once and can be then re-used.



(a) The optimal solutions (b) The solution #2

Fig. 4. The results of the CATMOS suite

The results of running the CATMOS are shown in Fig. 4. Fig. 4(a) shows all the possible optimal solutions for different budgets. The horizontal axis presents the total satisfaction that each solution offers, while the vertical represents, the cost of implementing each solution. Based on the budget, clients can pick the most appropriate solution knowing that this set of requirements is the optimal for the amount of money they want to spent. An example is given in Fig. 4(b). In this solution, the client can have the FR2 (Home Page), FR3 (Contact Us) and FR5 (Restaurant Presentation) requirements developed, which is the optimal set of requirements for a budget of 1100 units of money.

5.3 Code and Test Cases Generation

Code and test cases can be generated from the drawings. In this scenario, we use EGL to generate code for the structural and navigational parts of the described application. Code that implements the behavioural requirements (workflows) is not generated, yet. Plans for this process are presented in Section 6. In addition to that, we generate Selenium-Webdriver[7] test cases directly from the specification for the structural and navigational aspect of the application. The test cases can be used as primary artefacts for a Test-Driven Development approach where the developers create the structure and navigation from scratch.

6 Conclusions and Future Work

We presented a method that can be used to elicit and store requirements of applications. The method allows non-technical stakeholders to 'draw' the requirements with no previous training offering a low entry barrier for industrial use. It offers a flexible structure level to the requirements document (from completely un-structured to highly structured) without restricting the types of requirements that can be expressed. Finally, it can be used as a starting point for a number of MDE methodologies like code and test cases generation suites. It is highlighted though, that this approach could be beneficial in domains where the graphical representation of requirements adds value. In contrast, in domains where textual representations are more appropriate, the drawing overhead of the approach should be taken into account.

In the future, we aim to connect using M2M transformations, the requirements drawn using FlexRE with other MDE suites like WebRatio, which can perform full code generation.

Acknowledgments

This work was carried out in cooperation with Digital Lightspeed Solutions Ltd, and was supported by the EPSRC through the LSCITS initiative and part supported by the EU, through the MONDO FP7 STREP project (#611125).

References

1. Domberger, S., Hall, C., Li, E.A.L.: The determinants of price and quality in competitively tendered contracts. The Economic Journal (1995) 1454–1470

[7] http://docs.seleniumhq.org/projects/webdriver/

2. Abbott, B.: Requirements set the mark. Infoworld (2001) 45–46
3. Epner, M.: Poor project management number-one problem of outsourced e-projects. Research Briefs, Cutter Consortium **7** (2000)
4. Lowe, D.: Web system requirements: an overview. Requirements Engineering **8**(2) (2003) 102–113
5. Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Marshall, M.S.: Graphml progress report structural layer proposal. In: Graph Drawing, Springer (2002) 501–512
6. Kolovos, D.S., Matragkas, N., Rodríguez, H.H., Paige, R.F.: Programmatic muddle management. XM 2013–Extreme Modeling Workshop (2013) 2
7. Fernandes, K.J.: Interactive situation modelling in knowledge-intensive domains. International Journal of Business Information Systems **4**(1) (2009) 25–46
8. Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): a modeling language for designing web sites. Computer Networks **33**(1) (2000) 137–157
9. Ossher, H., Bellamy, R., Simmonds, I., Amid, D., Anaby-Tavor, A., Callery, M., Desmond, M., de Vries, J., Fisher, A., Krasikov, S.: Flexible modeling tools for pre-requirements analysis: conceptual architecture and research challenges. ACM Sigplan Notices **45**(10) (2010) 848–864
10. Kaindl, H., Smialek, M., Svetinovic, D., Ambroziewicz, A., Bojarski, J., Nowakowski, W., Straszak, T., Schwarz, H., Bildhauer, D., Brogan, J., et al.: Requirements Specification Language Definition: Defining the ReDSeeDS languages. Institute of Computer Technology, Vienna University of Technology (2007)
11. Escalona, M., Koch, N.: Metamodeling the requirements of web systems. Web Information Systems and Technologies (2007) 267–280
12. Jose Escalona, M., Aragon, G.: NDT. a model-driven approach for web requirements. Software Engineering, IEEE Transactions on **34**(3) (may-june 2008) 377 –390
13. Molina, F., Pardillo, J., Toval, A.: Modelling web-based systems requirements using WRM. In Hartmann, S., Zhou, X., Kirchberg, M., eds.: Web Information Systems Engineering WISE 2008 Workshops. Volume 5176 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2008) 122–131
14. Caine, S.H., Gordon, E.K.: PDL: a tool for software design. In: Proceedings of the May 19-22, 1975, national computer conference and exposition. AFIPS '75, New York, NY, USA, ACM (1975) 271–276
15. Ross, D., Schoman, K.E., J.: Structured analysis for requirements definition. Software Engineering, IEEE Transactions on **SE-3**(1) (jan. 1977) 6 – 15
16. Sommerville, I.: Software Engineering. 6th Edition. Addison Wesley Publishing Company Inc, Essex, England (2001)
17. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.: The design of a conceptual framework and technical infrastructure for model management language engineering. In: Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on, IEEE (2009) 162–171
18. Burton, F.R., Paige, R.F., Rose, L.M., Kolovos, D.S., Poulding, S., Smith, S.: Solving acquisition problems using model-driven engineering. In: Modelling Foundations and Applications. Springer (2012) 428–443

Taxonomy of Flexible Linguistic Commitments

Vadim Zaytsev

Universiteit van Amsterdam, The Netherlands, vadim@grammarware.net

Abstract. Beside strict linguistic commitments of models to metamodels, documents to schemata, programs to grammars and activities to protocols, we often require or crave more flexible commitments. Some of such types of flexible commitments are well-understood and even formalised, others are considered harmful. In this paper, we make an attempt to classify these types from the language theory point of view, provide concrete examples from software language engineering for each of them and estimate usefulness and possible harm.

1 Introduction

In software language engineering people often speak of linguistic commitment, structural commitment or commitment to grammatical structure [4]. Examples include:

- well-formed schema-less XML documents that commit to containing tags with allowed names and in proper combinations;
- XML documents that commit to the structure explicitly specified in the DTD or XSD;
- models that conform to their metamodel;
- programs in high level programming languages that commit to the structure specified by a grammar of such language and by other constraints of the designated compiler;
- programs that rely on some library and have to commit to calling its functions in proper order and with properly typed and ordered arguments;
- communicating entities that commit to sending and receiving messages according to the chosen protocol and must respect it in order to achieve compatibility and interoperability.

Any form of flexibility in such commitments is either not present or not considered. In this paper we propose to model it explicitly and classify its forms into several categories, varying in usefulness and the ability to lead to robust systems or to catastrophes.

The paper is organised as follows: §2 state the problem more clearly and formally. Based on that, §3 introduces three kinds of flexible language variations. Then, §4 classifies and explains all flexibly committing mappings within the framework. §5 proposes five kinds of streamlining helper mappings around the same language. §6 gives preliminary outlook at higher order manipulations such as composition of mappings,extension/restriction of them and constructing inverse functions. §7 concludes the paper.

2 Problem Statement

Consider a mapping $f : L_1 \to L_2$ which domain is a software language L_1 and which codomain is a software language L_2 (they can be the same in many cases, but let us look at the general case). Following the principle of least surprise, we could assume that f is surjective and total (i.e., its image fully coincides with its codomain and its preimage with its domain) so that it maps every element of L_1 to some element of L_2 and that every element of L_2 is reachable. By *flexible linguistic commitment* we will understand a situation when this expectation is violated.

3 Variations in Software Languages

Consider three variants based on a language L:

- By $\check{L}$ we will denote a language that is strictly smaller than L: it has more constraints and less elements. Committing to a subset of a language is in general not that harmful and means limited applicability of the tool or a mapping.
- By $\hat{L}$ we will denote a similarly strict superset of L which has then more elements and less constraints. Committing to a bigger language than intended is considered good practice in some areas that value robustness highly [7]. However, here we consider only "accidental" violations that extend the intended language in a way that preserves the semantics of the language instances up to heuristics that make sense for the problem domain.
- By $\tilde{L}$ we will denote another superset of L that refers to semantic changes — instances from $\tilde{L}$ are not just outside L, but they also allow interpretations that are incorrect according to the semantics of L. The existence of $\tilde{L}$ is the main cause for critique on robustness guidelines [1].

In order to stay universally applicable, we consider all definitions to be specific to the problem domain of the software language (without loss of generality, so called general purpose programming and modelling languages are assumed to belong to problem domains of Turing-complete algorithms and the everything-is-a-model paradigm respectfully). For example, unclosed tags are indicators of $\widehat{HTML}$ since the HTML language is meant to be processed in a soft and extremely flexible way, yet they are indicators of $\widetilde{XML}$ in most cases except for the absolutely trivial ones — say, one could argue that an XML document which is well-formed except one place with the construction like `<a><b></a>` is in $\widehat{XML}$ since it can be treated as `<a><b/></a>`. However, even `<a><b>c</a>` is already in $\widetilde{XML}$ since it has two intuitively good resolutions: `<a><b/>c</a>` and `<a><b>c</b></a>`. Yet, in a subdomain of XML that deals with exclusively empty or complex elements but never with cdata or mixed content, `<a><b>c</a>` would also belong to $\widehat{XML'}$.

2

	mapping from			
mapping to	$\check{L}_1 \to \dots$	$L_1 \to \dots$	$\hat{L}_1 \to \dots$	$\tilde{L}_1 \to \dots$
$\dots \to \check{L}_2$	correct program wrong language	non-surjectivity conservativeness	robustness	overrobustness
$\dots \to L_2$	partial applicability	perfect assumed	fault recovery	overrecovery
$\dots \to \hat{L}_2$	antirobustness	liberality	fault tolerance	overtolerance semirecovery
$\dots \to \tilde{L}_2$	fault introduction		shotgun effect	linguistic ignorance

Table 1. Forms of linguistic commitments of mappings with a domain L_1 and codomain L_2, classified by their preimages $\check{L}_1 \subset L_1 \subset \hat{L}_1$ and images $\check{L}_2 \subset L_2 \subset \hat{L}_2$. Additionally, $\tilde{L}$ is a special case where $\tilde{L}_i \supset L_i$, but the difference $\tilde{L}_1 \setminus L_1$ is unanticipated and $\tilde{L}_2 \setminus L_2$ is unintended.

4 Variations in First Order Mappings

Table 1 presents all possible preimage/image combinations. Let us consider each of them in turn more closely.

- ◇ $f : \check{L}_1 \to \check{L}_2$ (correct program, wrong language)
  Some mappings that look like having flexible linguistic commitments, are in fact (possibly) correct mappings working on a different language than expected. This kind of "flexibility" is easily fixable by correcting the specification.
- ◇ $f : \check{L}_1 \to L_2$ (partial mapping, limited applicability)
  Theoretically this scenario corresponds to the notion of partial function. In practice it can be well disguised in a wrapper that extends f to $L_1 \setminus \check{L}_1$ to behave like an identity function. If this is not done, this kind of flexibility is not flexible at all: it actually means that in order to use f, one needs to normalise its inputs in a more strict way than documented. In certain cases it manifests itself as a works-on-my-machine antipattern when such undocumented constraints concern configuration management and deployment details.
- ◇ $f : \check{L}_1 \to \hat{L}_2$ (antirobustness, ungratefulness)
  If robustness (see below) is to expect less and provide more, then antirobustness is its exact opposite: the demands on the input in this case are higher than officially specified, and even when they are met, the output is ungratefully relaxed.
- ◇ $f : \check{L}_1 \to \tilde{L}_2$ (fault introduction)
  If antirobust mappings can damage syntactic conformity, this variant can also do semantic damage. Such a tool is flexible beyond reasonable, it is inherently faulty: while holding surprisingly high expectations on its inputs, it generates outputs that are outright wrong.
- ◇ $f : L_1 \to \check{L}_2$ (non-surjective mapping, conservativeness)
  Concervative mappings that transform a language instance into an instance

with an even stronger linguistic commitment, are not surjective. For cases of $L_1 = L_2$ they are called language reductions and represent a form of traditional normalisers without extra tolerance for input violations. Most code generators are conservative: they cover the entire input language but there exist possible target language programs that will never be generated. Classic canonical normal forms in grammarware and databases are also this kind of conservative mappings: they are proven to exist for all inputs and infer standardised provably equivalent counterparts of them. Formally, this is one of the best kinds of flexibility.

- ⋄ $f : L_1 \to L_2$ (assumed / perfect)
  We list this form of commitment for the sake of completeness, but in fact it represents no flexibility: this is the standard commitment to grammatical structure [4], more or less precisely defined and precisely obeyed.
- ⋄ $f : L_1 \to \hat{L}_2$ (liberality)
  In mathematics, a function is under no circumstances allowed to return a value outside its codomain, but from our point of view this variant is a conceptual sibling of the conservativeness discussed above which corresponds to the lack of surjectivity. For the case of $L_1 = L_2$ this is called language extension: and it may be intentional — consider a situation of a coupled transformation sequence representing language evolution (if the language is backwards compatible, it will only contain language preserving and extending operators) or some kind of refinement/enrichment plan that recognises patterns of language use and transforms them into constructs of some more sophisticated language.
- ⋄ $f : L_1 \to \tilde{L}_2$ (fault introduction)
  What classifies mappings of this category is the lack of anticipation: it was probably meant to be a $\check{L}_1 \to \check{L}_2$ mapping, and as it turns out, it does not work correctly on unanticipated instances from $L \setminus \check{L}$. One of the typical examples is refactoring engines that claim to cover some programming language but in fact work correctly only on its subset while yielding irregular results whenever certain concurrency or subtyping patterns are involved [3].
- ⋄ $f : \hat{L}_1 \to \check{L}_2$ (robustness)
  This is precisely the model of the "be conservative in what you do, be liberal in what you expect from others", also known as Postel's Robustness Principle [7], or at least its harmless implementations. The input language is extended to a safe superset of L_1, but the output language is limited to a subset of L_2. Many normalisers work this way, accepting mostly valid entities of a language and mapping them onto a strict subset of the same language. For example, BibSLEIGH [15], a project involving a large JSON database, has a normaliser that traverses all JSON files, including those that have trailing commas in lists or dictionaries, as well as those with naïve quoting, and transforms each of them into an LRJ file (Lexically Reliable JSON) which is basically valid JSON with extra guarantees of one key-value pair per line and lines being sorted lexicographically.
- ⋄ $f : \hat{L}_1 \to L_2$ (recovery)
  A mapping that accepts slightly more than obligatory yet remains true to its

output language, represents error recovery: in the simplest case of $L_1 = L_2$ this may be the only purpose of the mapping, but it does not have to be. For example, consider notation-parametric grammar recovery, a technique that takes a human-written error-containing language document and a definition of the format it is supposed to have, and yields a well-formed grammar extracted from it [12]. Most of its heuristics are such mappings. Approximate pattern matching [6] and semiparsing [13] also belong to the same group, as do screen-scraping libraries like Beautiful Soup [8].

- $f : \hat{L}_1 \to \hat{L}_2$ (tolerance)
  In the terminology of negotiated mappings, the previous kind of flexible commitment represented adaptation through adjustment; this one is adaptation through tolerance [14]. Such a mapping still needs to cope with the unexpected outputs, but has an option of propagating the unexpected parts further down the pipeline instead of fixing them.
- $f : \hat{L}_1 \to \tilde{L}_2$ (shotgun effect)
  This kind of mapping has been identified in the technological space of internet security and named "shotgun parsing" there [2]. A shotgun parser is a program that is meant to check its input for linguistic commitment and sanitise it, yet in the interest of performance optimisations does not perform a thorough check and limits itself to fundamentally flawed approaches such as regular means of treating context-free languages or using plain string substitution for escaping; as a result, the system becomes vulnerable to malevolent input (comment bombs, SQL injections, etc). Every shotgun parser in the pipeline increases the span of possible treatments of data, hence the shotgun metaphor.
- $f : \tilde{L}_1 \to \check{L}_2$ (overrobustness)
  A few paragraphs above we reintroduced robustness as input type contravariance and output type covariance. Overrobustness does the same but crosses the syntax-semantics border and hence becomes dangerously ambiguous, nondeterministic and in general error-prone. Many examples of overrobustness leading to security bugs were given by Meredith Patterson et al. at `http://langsec.org` [2,9].
- $f : \tilde{L}_1 \to L_2$ (overrecovery)
  Overrecovery is a process of applying heuristic-based fixes to semantically incorrect language instances with the goal to return to the intended output language. Some aggressive heuristics like parenthesis matching from notation-parametric grammar recovery [12] fall into this category, as well as desperate matches in semiparsing [13].
- $f : \tilde{L}_1 \to \hat{L}_2$ (overtolerance, semirecovery)
  Overtolerance is a form of harmful error handling when semantic errors in the input are presumably fixed, but the output is still not always guaranteed to be syntactically correct.
- $f : \tilde{L}_1 \to \tilde{L}_2$ (ignorance)
  The ultimate form of flexible linguistic commitment is complete linguistic ignorance: our inputs can be broken beyond any hope of unambiguous repair, and the outputs are not much better in that respect. All lexical analysis

	$L \to \dots$	$\hat{L} \to \dots$	$\tilde{L} \to \dots$
$\dots \to \check{L}$	$\overline{\vee}$ *canoniser*	$\overset{\wedge}{\vee}$ *normaliser*	$\overset{\sim}{\vee}$ *regulator*
$\dots \to L$	id	$\overset{\wedge}{-}$ *codifier*	$\simeq$ *calibrator*

Table 2. Symbols and names for streamlining mappings.

methods belong to this category and are still being often use due to their extreme speed, ease of development and virtually unlimited flexibility, despite limited expressiveness and abundant false positives and negatives. Some migration and analysis projects of considerable size have been reported being completed with language ignorant lexical methods [5,10].

5 Streamlining Mappings

Before we try to compose flexibly committed mappings and consider higher order combinators, we need to introduce a special subcategory of streamlining mappings that help us "get back" to L or even $\check{L}$ whenever we deviate. In particular, we have a need for the following five (summarised on Table 2:

- *Canoniser* ($\overline{\vee} : L \to \check{L}$) has a normal (precise) commitment to L and produces a strict subset of it. Typically this is some kind of canonical normal form that makes sense for the chosen technological space; if it is strictly canonical, there is usually a proof of its uniqueness up to L.
- *Codifier* ($\overset{\wedge}{-} : \hat{L} \to L$) is flexible with its input because it applies certain rules for recovery which are applied until the output conforms to all expectations of L.
- *Normaliser* ($\overset{\wedge}{\vee} : \hat{L} \to \check{L}$) implements a classic Postel-style normalisation scheme that we have briefly discussed before: it is liberal with respect to its input and conservative with respect to its output. In our formalisation it also means that $\overset{\wedge}{\vee} \equiv \overline{\vee} \circ \overset{\wedge}{-}$ (a normaliser is equivalent to the superposition of a codifier and a canoniser), and indeed, any normaliser can be conceptually studied as a composition of two parts implementing the liberality and the conservativeness accordingly. We will consider superposition of flexibly committing functions in the next section in more detail.
- *Calibrators* ($\simeq : \tilde{L} \to L$) and *regulators* ($\overset{\sim}{\vee} : \tilde{L} \to \check{L}$) can be decomposed similarly in various ways, but the key when considering them is remembering that the main distinction between $\hat{L}$ and $\tilde{L}$ is that $\hat{L} \setminus L$ is anticipated and $\tilde{L} \setminus L$ is not. Hence, when something is not anticipated, it cannot be casually accounted for by an automated streamliner. In practice developing calibrators and regulators requires the same steps as developing codifiers and normalisers, with additional effort dedicated to testing and documenting recovery heuristics, which are usually unreliable.

$f \circledcirc g$		g, applied first			
		$\ldots \rightarrow \check{L}$	$\ldots \rightarrow L$	$\ldots \rightarrow \hat{L}$	$\ldots \rightarrow \tilde{L}$
f, applied second	$\check{L}' \rightarrow \ldots$	$f \circ g$, if $\check{L} \subseteq \check{L}'$ $f \circ \overset{-}{\vee} \circ g$	$f \circ \overset{-}{\vee} \circ g$	$f \circ \underset{\vee}{\wedge} \circ g$	$f \circ \underset{\vee}{\sim} \circ g$
	$L \rightarrow \ldots$	$f \circ g$	$f \circ g$	$f \circ \underset{-}{\wedge} \circ g$	$f \circ \simeq \circ g$
	$\hat{L}' \rightarrow \ldots$	$f \circ g$	$f \circ g$	$f \circ g$, if $\hat{L} \subseteq \hat{L}'$ $f \circ \underset{-}{\wedge} \circ g$	$f \circ \simeq \circ g$
	$\tilde{L}' \rightarrow \ldots$	$f \circ g$	$f \circ g$	$f \circ \underset{-}{\wedge} \circ g$	$f \circ \simeq \circ g$

Table 3. Superposition of two mappings with flexible commitments ($f \circledcirc g$) expressed as normal superposition of f, g and possibly some streamliners. Dashes simply denote non-uniqueness: for example, $\hat{L}$ may be a *different* superset of L than $\hat{L}'$.

6 Manipulating Flexible Mappings

6.1 Extension and Restriction

For every possible combination of input and output types (strictly speaking, for any input type since output type is irrelevant), a restriction on L is defined straightforwardly as a function which returns whatever the original function would have returned, for all inputs from the restricted set, and is underfined otherwise. An extension of a function deliberately committing to a subset of the intended language, on the other hand, cannot be defined in a general fashion. Hence, for any f which preimage is L, $\hat{L}$ or $\tilde{L}$, we get a $f|_L$ for free, but the extension $f|_L$ for $f : \check{L} \rightarrow L'$ is, generally speaking, unknown.

A more interesting observation is that restriction can affect the flexibility of the linguistic commitment to the output language as well: for example, if $f : \hat{L}_1 \rightarrow L_2$, then $f|_{L_1} : L \rightarrow L_2$ or possibly $f|_{L_1} : L \rightarrow \check{L}_2$. This has bad consequences on calculating inverses of restrictions.

6.2 Inverse Functions

Due to the nature of our formalisation that considers preimages and images instead of domains and codomains, all functions that map between L_1, $\check{L}_1$, $\hat{L}_1$ and L_2, $\check{L}_2$, $\hat{L}_2$, have straightforward inverse mappings. Their non-unique existence is guaranteed (the proof is a direct consequence of the definitions of a premiga and an image). For example, for any $f : \check{L}_1 \rightarrow \hat{L}_2$ there is at least one $f^{-1} : \hat{L}_2 \rightarrow \check{L}_1$. Inverses of restrictions can be more desirable, but less reliable, since there is no guarantee that for an arbitrary $f : \hat{L_1} \rightarrow \hat{L_2}$, the image of $f|_{L_1}(L_1) \cap L_2 \neq \varnothing$.

6.3 Composition

Table 3 summarises the resulting superpositions of two flexible mappings with respect to all possible kinds of mappings. The most obvious case is the following:

$$\circledcirc : (L_2 \rightarrow L_3) \rightarrow (L_1 \rightarrow L_2) \rightarrow (L_1 \rightarrow L_3)$$

However, the following compositions are also universally safe (variations of L_2 correspond to variations of L in the table):

$$\circledcirc : (L_2 \to L_3) \to (L_1 \to \check{L}_2) \to (L_1 \to L_3)$$
$$\circledcirc : (\hat{L}_2 \to L_3) \to (L_1 \to \check{L}_2) \to (L_1 \to L_3)$$
$$\circledcirc : (\tilde{L}_2 \to L_3) \to (L_1 \to \check{L}_2) \to (L_1 \to L_3)$$
$$\circledcirc : (\hat{L}_2 \to L_3) \to (L_1 \to L_2) \to (L_1 \to L_3)$$
$$\circledcirc : (\tilde{L}_2 \to L_3) \to (L_1 \to L_2) \to (L_1 \to L_3)$$

The result type of such compositions given above is a possible overapproximation, because technically we combine g with $f|_{L_2}$ or $f|_{\check{L}_2}$, not with f itself, so it is possible for $f \circledcirc g$ with the codomain L_3 to have image $\check{L}_3$.

The rest often require streamliners, with two special cases stemming from the fact that in our notation $\hat{L}$ expresses *any* superset of L and not a particular one. Thus, it can be the case that $\check{L}_2 \subseteq \check{L}'_2$ and then $\circledcirc : (\check{L}'_2 \to L_3) \to (L_1 \to \check{L}_2) \to (L_1 \to L_3)$ is expressible with a simple superposition: $f \circledcirc g = f \circ g$, but otherwise we need an $\check{L}'_2$-canoniser to glue them together: $f \circledcirc g = f \circ \overline{\vee} \circ g$. Similarly to the safe case discussed above, the image of such composition can be $\check{L}_3$ if $\check{L}_2 \neq \check{L}'_2$.

7 Conclusion

This paper is an attempt to investigate Postel's Robustness Principle ("be conservative in what you do, be liberal in what you expect from others") [7,1], in particular to follow the Postel's Principle Patch ("be definite about what you accept") [9] and the formal language theoretical approach to computer (in)security, in the general software language engineering view (not limited to internet protocols and even data languages). The result was a taxonomy of several families of language mappings depending on the inclusion relations between their domains and preimages, as well as codomains and images. The taxonomy (Table 1) contains two forms of precise commitment and several forms of harmful and profitable flexible commitments. Streamlining mappings (Table 2), mapping superpositions (Table 3) and other details of manipulating mappings with such flexible commitments, have also been considered. This classification is a refinement with respect to any previously existing approach, and provides means to identify safe combinations of mappings and streamliners. Formal treatment of tolerance [11] remains future work, and in general a categorical approach should provide even more solid foundation than a set theoretical one, which can also be refined further to yield interesting and useful properties.

References

1. E. Allman. The Robustness Principle Reconsidered. *Communications of the ACM*, 54(8):40–45, Aug. 2011.
2. S. Bratus and M. L. Patterson. Shotgun Parsers in the Cross-hairs. In *Brucon*, 2012. http://langsec.org.
3. M. Gouseti. A General Framework for Concurrency Aware Refactorings. Master's thesis, UvA, Mar. 2015. http://dare.uva.nl/en/scriptie/502196.
4. P. Klint, R. Lämmel, and C. Verhoef. Toward an Engineering Discipline for Grammarware. *ACM ToSEM*, 14(3):331–380, 2005.
5. S. Klusener, R. Lämmel, and C. Verhoef. Architectural Modifications to Deployed Software. *Science of Computer Programming*, 54:143–211, 2005.
6. V. Laurikari. TRE. http://github.com/laurikari/tre, 2005.
7. J. Postel. DoD Standard Internet Protocol. RFC 0760, 1980.
8. L. Richardson. Beautiful Soup. http://www.crummy.com/software/BeautifulSoup, 2012.
9. L. Sassaman, M. L. Patterson, and S. Bratus. A Patch for Postel's Robustness Principle. *IEEE Security and Privacy*, 10(2):87–91, Mar. 2012.
10. D. Spinellis. Differential Debugging. *IEEE Software*, 30(5):19–21, 2013.
11. P. Stevens. Bidirectionally Tolerating Inconsistency: Partial Transformations. In *FASE*, volume 8411 of *LNCS*, pages 32–46. Springer, 2014.
12. V. Zaytsev. Notation-Parametric Grammar Recovery. In A. Sloane and S. Andova, editors, *LDTA*. ACM DL, June 2012.
13. V. Zaytsev. Formal Foundations for Semi-parsing. In S. Demeyer, D. Binkley, and F. Ricca, editors, *CSMR-WCRE ERA*, pages 313–317. IEEE, Feb. 2014.
14. V. Zaytsev. Negotiated Grammar Evolution. *JOT*, 13(3):1:1–22, July 2014.
15. V. Zaytsev. BibSLEIGH: Bibliography of Software Language Engineering in Generated Hypertext. In A. H. Bagge, editor, *SATToSE*, pages 59–62, July 2015.

Prototizer: Agile on Steroids

Aram Hovsepyan[1], Dimitri Van Landuyt[2]

[1] CODIFIC
aram@codific.eu
[2] iMinds-DistriNet KULeuven
dimitri.vanlanduyt@cs.kuleuven.be

Abstract. The model-driven software development (MDSD) vision has booked significant advances in the past decades. MDSD was said to be very promising in tackling the "wicked" problems of software engineering in general. However, a decade later MDSD is still far from becoming widely recognized within the mainstream software development. At the same time Agile software development methodologies are widely considered as the way to go. This is counter-intuitive as MDSD seems to be the right methodology to boost Agile approaches. From Agile software development perspective, design models are a waste.
In this experience report, we present Prototizer, a tool based on model-driven software engineering that could boost the Agile vision. We present a validation of Prototizer on a recent case study and discuss the main lessons learned throughout the past years.

Keywords: agile software development, minimum viable product, model-driven development process, prototizer, lean entrepreneurship

1 Introduction

Given the advances in hardware technology, software development in general is becoming an increasingly complex activity. For about a decade the model-driven software development (MDSD) vision has seemed very promising in efficiently tackling the essential complexities of the software development process [1]. The MDSD vision, primarily focused on the vertical separation of concerns, aims at reducing the gap between problem and software implementation domains through the use of models that describe complex systems at different abstraction levels and from a variety of perspectives. Automated code generation, claimed to be one of the most valuable assets of MDSD, allows us to translate these models instantaneously into code. Strangely, MDSD is still far from becoming accepted within the mainstream software development [2].

As opposed to MDSD, Agile methodologies are currently considered to be the best practices within software development. The spectrum of agile methodologies is very broad. The core principles are typically focused on fast iteration cycles, responsiveness to change, early customer involvement, "good design", etc. Agile approaches aim at reducing the waste of big up-front analysis, planning and documentation. Nonetheless, even the most code-centric agile methodologies suggest the use and evolution of design models for communication and documentation purposes. From this perspective, it is our strong belief that MDSD is a necessary building block within a mature agile methodology. Introducing a lightweight MDSD within Agile methodologies could further reduce the waste, enforce a good design, aid with rapid prototyping and increase customer involvement. Indeed, if we use models for representing software design, we might as well use them as software artefacts, rather than merely a communication and documentation tool.

In this experience report, we present Prototizer, an MDSD supporting tool that could be used within any Agile software development process. Prototizer embraces only a small part of the MDSD philosophy, namely using models as first-class citizens and generating partial code from these models. Nevertheless, this small part is the enabler of the true Agile vision. Over the course of the past five years, we have developed more than ten different bespoke CRM/ERP web-based software systems using Prototizer. More importantly, we are still maintaining and expanding these software systems using Prototizer. In this paper, we present one of these systems.

The remainder of the paper is structured as follows. In section 2, we provide essential background information on agile software development and model driven software development. We also describe the problem statement in detail. In sections 3 and 4, we present our solution and its application on a recent case-study. Section 5 provides a brief evaluation of Prototizer. Finally, section 6 concludes this paper.

2 Agile vs. Models

This section provides a brief overview of agile and MDSD methodologies by focussing on the key concepts that are in the core of the two paradigms.

2.1 Agile Software Development

Agile software development is a rather overloaded term and many existing approaches claim to be agile. The most well-known approaches in literature include eXtreme Programming (XP) [3], Scrum [4], Feature Driven Development (FDD) [5], Kanban [6], Dynamic systems development method [7], DevOps [8]. Despite their intrinsic differences, which are out of the scope for this paper, all agile approaches are focused on concepts such as adaptive planning, evolutionary development, early delivery, continuous improvement, etc. More importantly, agile has become a synonym of lean, i.e., reducing waste and focusing on efficiency. This means that the heavyweight planning, documentation, software architecture and design phases are reduced significantly. Fast iteration cycles (referred to as timeboxes or sprints) with a duration of weeks or even days resulting in a completely implemented, validated and verified subset of requirements is a central theme in all agile approaches [7]. Nonetheless, even the most code-centric agile approaches advocate the use of design models [3]. While models are used for documentation and communication purposes, established agile approaches advice the use of UML and do not exclude the use of "complex models using specific notations" [7]. It is our strong belief that a lightweight MDSD process can substantially increase the added value of agile approaches. Keeping models up-to-date requires a substantial rigor and therefore is highly challenging in an agile context. In our view, fully automated, but partial code generation will provide instant prototyping highly praised within agile.

2.2 Models

Similar to agile, model-driven software development (MDSD) also covers a relatively broad spectrum of ideas, techniques and tools. Despite their differences, we believe two key ideas are essential within most MDSD approaches.

Code generation Code generation is a central selling point behind MDSD [9]. Code generation is instantaneous and saves time for the developers[3]. Code generation improves the source code quality as generated code can be tailored to follow best coding practices and can be considered to be bug-free [9]. Finally, this instant code generation enables developers to play with the solution and quickly deliver prototypes of the final system. Fast prototyping enables early validation that is a central theme in agile approaches.

Models as primary software development artefacts The models within a MDSD approach are no longer a mere piece of documentation, but actually an essential software development artefact. Indeed, models have to be precise and complete as they are fed to a code generator. As a result, models are always up-to-date with the source code that makes them a valuable "lingua franca" between the stakeholders. Software systems developed using MDSD are less likely to evolve to a spaghetti-like systems where only the developers can manage to find their way [9].

2.3 Problem Statement

It is our strong belief that there are two key obstacles that prevent MDSD from entering the mainstream software development and supporting the currently prevalent agile software development processes.

Rigid Code Generators. Typically, in a MDSD approach the code is generated based on a template (e.g., Eclipse JET) or a script that is a programming language on its own (e.g., Acceleo [10]). One of the essential problems in MDSD

[3] Throughout this work we only consider the MDSD vision where the code generation is partial. Concretely, this means that the developers will typically generate the overall structure and the behaviour will be manually programmed by the developer.

is that these templates/scripts are claimed to be reusable. In extreme cases, out of the box generators are created and published, such as "THE Java code generator", "THE C code generator", etc. It is unlikely that two different software development firms will be happy with the same generator. Unfortunately, the existing generators are often too rigid. It could be very challenging to quickly edit the already messy and complex code generation templates. Finally, code generators must support iterative development, hence, the manually written code should not be rewritten by the automated code generation. Although MDSD research has always stressed the importance of this issue, its solution is far from trivial. The concept of a "protected code section" seems to solve the problem at a first glance, however it is not clear how to properly use them.

Steep Learning Curve. Even the most simple MDSD approach has a rather steep learning curve. We believe that the main cause is that MDSD approaches often try to oversell and become too heavyweight [9]. For a developer who is new to MDSD and only has a rather superfluous understanding of UML class diagrams it would be extremely difficult to join the models club. There is a lack of simple lightweight MDSD success stories the potential followers could start playing with. Once again, the technology providers typically try to provide ready-to-use generators, rather than focusing on the mechanisms on how to modify the existing generators or create new generators. As a result, even the early adopters are unable to step into the world only the technology providers understand.

In the next section, as early adopter we present our toolset that we have created based on existing MDSD technologies.

3 Prototizer

We refer to Prototizer as the toolset that enables the model-driven and agile software development process. In this section we describe both the process we follow as well as the toolset itself.

3.1 Prototizer Software Development Process and Toolchain

Figure 1 presents the development process showing each development activity along with their structural connections to other activities. The solid lines on the figure are both workflow and artefact transitions from one development activity to another. The dashed lines represent traceability links between different artefacts. Traceability information currently falls out of the scope of our approach and will not be discussed in this paper. The presented process process is in line with the V-Model. We briefly describe each of the development phases, along with the underlying technology that we have used.

Requirements Analysis This phase refers to both business requirements analysis as well as their translation into the technical requirements analysis. This activity is done using a more traditional approach, i.e., by using a text editor.

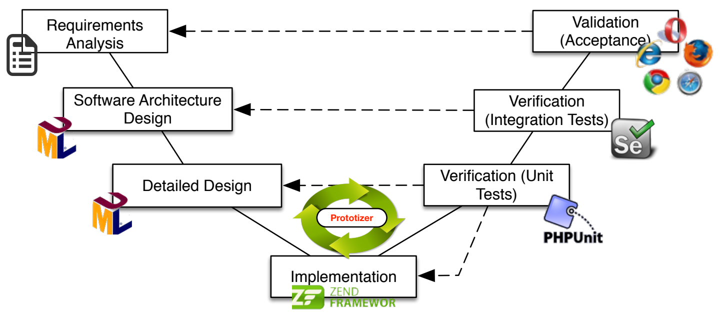


Fig. 1. Prototizer Software Development Process

Software Architecture Design This phase defines the architecture of the overall software system. The software architecture is created in UML by the means of component/connector and deployment diagrams. Currently, we do not leverage the software architecture explicitly in the code generation.

Detailed Design This phase describes the detailed UML class diagram that is further used for code generation.

Implementation Our firm mainly leverages the PHP Zend Framework as an underlying platform. However, virtually any programming language and platform can be used for the implementation.

Verification (Unit and Integration Tests) The verification phase focuses on automated unit and integration tests that are an essential part of any systematic software development process. Given the PHP implementation platform, we further rely on PHPUnit and Selenium WebDriver for the unit testing and integration testing.

Validation Finally, the end customer is expected to perform the validation of the product release and officially accept the release. This is done by using a modern web-browser.

The V-model is traditionally not considered to be agile, as it represents an extension of the waterfall model. Thus, to improve the dynamics of this process we leverage the Dynamic System Development Method (DSDM) Atern agile project delivery framework used for software development [7]. The idea behind DSDM is to develop a solution iteratively starting from global view of the product. Figure 2 presents the timebox concept that is a key technique in DSDM Atern. It represents the iterative process to control the creation of the product under development with specific review points to ensure the quality of the product and the efficiency of the delivery process. A more detailed description of DSDM Atern is out of scope for this paper [7].

3.2 Prototizer Tool

Prototizer is an open-source tool implemented as an Eclipse plug-in and can be downloaded from [11]. Prototizer is largely based on MOFScript that is an

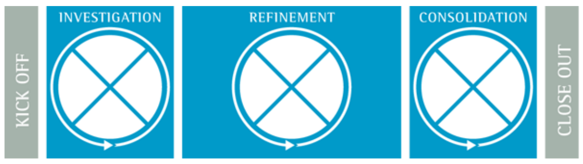


Fig. 2. Timebox

open-source code generation technology developed by SINTEF [12]. Prototizer transforms the input UML model into code based on a user-specified pluggable and extensible cartridge. Over the years we have developed over five different generation cartridges. Each generation cartridge contains two components, i.e., resource copier and generation script.

Resource Copier The resource copier simply copies various static resources, such as libraries, Javascript/HTML/CSS files, into the file structure of the project. The set of resources can be easily manipulated by the developer by simply managing the static files within the plugin cartridge folder. These resources are typically specific for a certain company or even project domain.

Generation Scripts The generation scripts are used by Prototizer to translate the UML model into code. Modifying a generation script is straightforward as MOFScript is an imperative language syntactically similar to Java. For the specifics of the generation scripts we refer to the MOFScript specification [12]. We leverage two complementary techniques in order to make the code generation scripts sufficiently flexible when it comes to manual code refinements.

1. We use protected code sections that are placeholders for manually refined source code that are kept intact upon subsequent generation steps.
2. In certain cases, protected code sections could place unnecessary constraints on the manual coding. In order to overcome this problem, we leverage the generation gap pattern [13]. The generated code is placed in abstract superclasses that can be easily subclassed with manually written code.

4 Case study: CODIFIX

The case study presented in this paper is a simplified version of our own enterprise resource planning system named CODIFIX. CODIFIX initially consisted of a rather primitive content management system for our website. However, we have gradually added various new modules that have introduced a substantial set of new functionalities. In this section, we will briefly describe two of the CODIFIX modules, i.e., the content management system and the issue tracking system. We focus mainly on the models from which the source code is continuously and incrementally generated. At the end of this section, we provide an overview of the artefacts that are actually generated from the design models.

4.1 Technology Stack

CODIFIX is implemented in PHP by leveraging the Zend Framework version 2. Note that we do not use the Doctrine framework that provides a transparent database storage. Rather, the generation step creates the complete database API ready to use by the developers. We also rely on client-side functionality written by third parties in Javascript. As an underlying database we use MySQL.

4.2 Content Management System

Figure 3, presents the class diagram of the simple content management system (CMS) model we have designed to use for our informative website. The CMS consists of menus (*Menu*) denoting the pages. Each menu has a specific language (*Language*) and can have a parent menu. The website information is represented as contents (*Content*) where each content can belong to a menu object. The attributes of the classes and the semantics that are assigned to the model elements and used in the generation are out of scope for this paper.

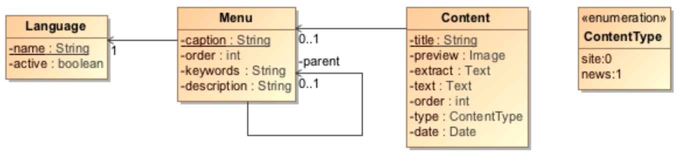


Fig. 3. Content Management System

4.3 Issue Tracking System

Figures 4 presents the diagram of the issue tracking system model (ITS). The ITS groups tasks (*Task*) in projects (*Project*). Additional task information is represented by attachments (*Attachment*) and comments (*Comment*). The task and project related information is obviously linked to users (*User*) each of whom belongs to a certain client (*Client*). The access control is currently hard-coded and depends on the linking between *User*, *Role* and *System.*

4.4 Generated artefacts

Database scheme and API The complete data layer as well as the communication API is generated by Prototizer. The database scheme is generated as an .sql dump. We have developed a simple scripting mechanism to synchronise the generated database scheme with the actual running database. The database communication API is a collection of classes that allows systematic manipulation of the database entries for each of the classes within the UML models.

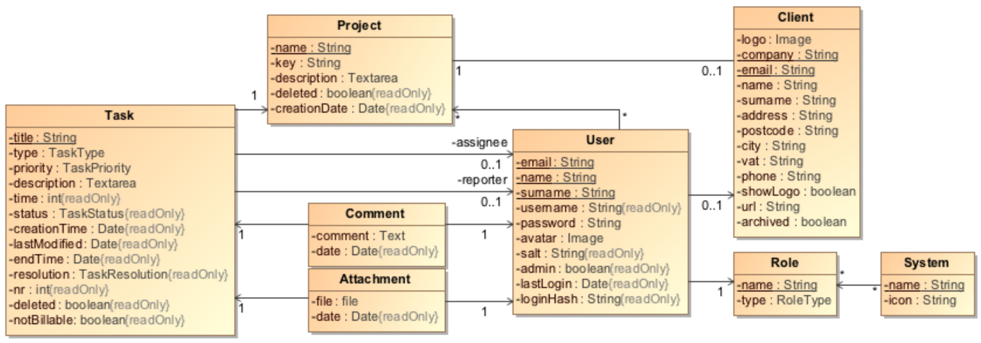


Fig. 4. Issue Tracking System

We currently do not leverage on frameworks like Doctrine (the PHP version of Hibernate) and the database communication API represents a relatively large codebase. Typically, this part of the generated code is never modified manually as the UML model within the Prototizer philosophy is semantically complete.

Model classes Each modelling entity (along with its relationships) is translated into a corresponding *PHP Class*. These classes represent the models in MVC terminology. Obviously, the classes have pre-generated list of attributes as well as getter and setter functions. In addition, we also generate validation rules for each attribute as defined by its type. For instance, the *order* attribute in *Menu* class must be an integer. The model classes are typically manually refined, hence, protected code sections denote the places where this can be realised. In general, the attribute information as well as their getters and setters should never be modified manually. On the other hand, methods like *toString()* and any additional methods that are added by the developer are protected by Prototizer.

Controllers and views The basic features for most of our systems (including CODIFIX) are the web-based create, retrieve, update and delete (CRUD) interfaces to manipulate the database entities. These interfaces are also generated by Prototizer. In MVC terminology these are the controllers and the views. The controllers are in charge of processing the HTTP requests and constructing the HTML views that are sent back to the web-browsers by the web-server. The views are mainly HTML scripts with template parameters that are dynamically instantiated by the corresponding controllers.

Generated code percentage Overall, our currently running CODIFIX project contains 17546 lines of generated PHP/HTML code from a total codebase of 22281 lines of code (including comments and white spaces). Current best practices in Zend Framework would require one to write all this code manually. Nonetheless, the productivity increase is limited by two factors. Firstly, not all of the generated code is actually used. Given the ease of code generation we generate quite a few helper functions that are not always needed and used. Secondly, the generated code is by definition trivial as it constitutes the repetitive part of the code. However, writing the code manually when it can be generated is a 100% waste of time.

5 Lessons Learned

In the past five years we have iteratively developed numerous custom web-based CRM and ERP systems using Prototizer. All of these systems are still heavily used by our customers who often submit change requests. In this section, we provide an overview of the lessons learned within the context of our firm.

5.1 Benefits

The process behind Prototizer is in line with agile and virtually any agile framework can be plugged in as a concrete dynamics of the process. Prototizer is a clear realization of the MDSD vision regarding the centrality of models, instantaneously generated source code and the increased prototyping abilities. This saves a substantial amount of time and allows our developers to focus on core problems rather than spending time in typing code. The generated code is considered to be bug-free as we assume the generation cartridges are bug free. Prototizer forces the designers to create complete UML models of the system, hereby making the communication between designers and developers in the context of the system structure much more clear. Prototizer also enforces a specific code structure that is valuable especially for the less experienced developers within our firm. In our experience, the long-term benefits of Prototizer are even more critical. The existence of a complete and up-to-date UML model increases the system longevity and substantially reduces the maintenance cost. Within our firm, the cost of maintenance (e.g., new features, change requests) often by far exceeds the original cost of development. This is confirmed by various studies in the industry (e.g., [14]).

5.2 Drawbacks

Virtually any code generation approach introduces additional constraints for the developers. The protected sections where developers are expected to operate sometimes lead to two problems. Firstly, inexperienced developers inevitably misplace manually written code that leads to overwritten code on subsequent iterations. While the code is not really lost (thanks to version control), such situations involve an additional overhead. Secondly, certain manual refinements could be relatively complicated to fit within the protected sections. Developers are sometimes forced to duplicate code in order to achieve the desired result.

5.3 Evaluation

From a research point of view Prototizer is not innovative. In fact, the building blocks of Prototizer, i.e., MOFScript and EMF were stable almost a decade ago. However, from a state-of-the-practice point of view, Prototizer is a pragmatic answer to both problems presented in section 2.3. Generation cartridges can be easily modified and new cartridges can be quickly created by example. The resource copier requires simply moving around files and folders that are needed for a specific project type. MOFScript is a powerful, yet very simple language for creating and modifying code generation scripts. Prototizer is a very lightweight approach, only focusing on a very small subset of UML, i.e., class diagrams. Instead of focusing on a programming language (PHP code generator), we have created generation cartridges for a specific framework (Zend code generator). We believe that these aspects contribute to reducing the steepness of the MDSD learning curve. However, we have not validated either of these claims in a systematic fashion.

6 Conclusion

In this experience paper, we have presented Prototizer - a tool that enables a boosted agile software development approach. Prototizer, which is based on existing model-driven software engineering building blocks, enables the use of the design models as actual software development artefacts, rather than mere documentation.

References

1. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Proceedings of the 29th ICSE, IEEE Computer Society (2007) 37–54
2. Fieber, F., Regnat, N., Rumpe, B.: Assessing usability of model driven development in industrial projects. CoRR **abs/1409.6588** (2014)
3. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change (2Nd Edition). Addison-Wesley Professional (2004)
4. Schwaber, K., Beedle, M.: Agile Software Development with Scrum. 1st edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (2001)
5. Palmer, S.R., Felsing, M.: A Practical Guide to Feature-Driven Development. 1st edn. Pearson Education (2001)
6. Anderson, D.: Kanban. Blue Hole Press (2010)
7. DSDM Consortium: The DSDM Atern Handbook. DSDM Consortium (2008)
8. Httermann, M.: DevOps for Developers. 1st edn. Apress, Berkely, CA, USA (2012)
9. Frankel, D.: Model Driven Architecture: Applying MDA to Enterprise Computing. John Wiley & Sons, Inc., New York, NY, USA (2002)
10. Obeo: Acceleo. (http://www.eclipse.org/acceleo)
11. CODIFIC: Prototizer. (http://prototizer.com)
12. SINTEF: MOFScript. (http://modelbased.net/mofscript/)
13. Fowler, M.: Domain Specific Languages. 1st edn. Addison-Wesley Professional (2010)
14. Erlikh, L.: Leveraging legacy system dollars for e-business. IT Professional (2000)