Intelligent Code Generation for Model Driven Web Development

Emanuele Falzone and Carlo Bernaschina

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano — Piazza L. Da Vinci, 21. I-20133 Milano, Italy
`emanuele.falzone@mail.polimi.it, carlo.bernaschina@polimi.it`

Abstract. Model Driven Development requires proper tools to derive the implementation code from the application models. However, the use of code generation tools may interfere with the software development and maintenance practices, because most state-of-the art tools are incapable of preserving manual modifications to the code when the implementation is regenerated from the models. In this paper, we present an approach which organizes the model transformation rules and the code architecture in a way that preserves the parts of the code that are defined outside the model-and-generate cycle, such as the code defining the look and feel of the user interface and the connection to the required service endpoints.

Keywords: Model Driven Development, Code Generation, Agile Development

1 Introduction

Model Driven Development (MDD) is the branch of software engineering that advocates the use of *models*, i.e., abstract representations of a system, and of *model transformations* as key ingredients of software development. Developers use a general purpose (e.g. UML[1]) or domain specific (e.g. IFML[2]) modeling language to specify systems under one or more perspectives, and use (or build) suitable chains of *transformations* to refine the models into a final product (e.g. executable code). Abstraction is the far most important aspect of MDD. It enables developers to validate high level concepts and introduce details, which increase complexity, later in the process. In the *Forward Engineering* approach details are introduced via model transformations, which iteratively refine the model eventually getting to the final product. After each model change, the process is reiterated to produce a new version. Model to Model (M2M) and Model to Text (M2T) transformations are tailored to achieve this goal. Transformations can be grouped into two groups: **I) Model Enhancement**. The model is enhanced by the introduction of details that are fixed, or easily derivable from the model itself with fixed rules. An example is a M2M transformation in the Entity Relationship domain which maps hierarchies into equivalent entities and

[1] http://www.uml.org [2] http://www.omg.org/spec/IFML/1.0/

relationships. **II) Model Specialization**. The model is refined by introducing details specific for a target platform (e.g. system, language, . . .). This is obtained via a M2M or M2T transformation having a high-level source meta-model and a platform specific target meta-model. An example is a M2T transformation that maps a state machine representing GUI interactions to the source code of a specific GUI framework. In this work we focus on model specialization.

Transformations should map each valid source model into one valid target model deterministically. When the target meta-model has a higher expressive power two scenarios are possible: **I) Exploitation**. The transformation exploits the expressive power of the target meta-model to express in higher detail the concepts defined in the source model. An example is a M2M transformation that defines the semantics of a language by mapping it to a more expressive and well defined one, like in [7], where Statecharts are used to define the semantics of WebML, and in [5], where Place Chart Nets (PCN) are used to describe the semantics of IFML. If the target meta-model (Statecharts, PCN) is more expressive than the source model (WebML, IFML), all the valid models that cannot be produced have no practical usage for the specific purpose of the transformation. **II) Ambiguity**. More then one valid target model is a possible output candidate for the M2T transformation. The developer is responsible of directing the transformation to take a decision among the possible alternative outputs. An example is a M2T transformation that generates a GUI implementation from an application model [4]; many visual representation can be produced, by taking different assumptions on layout and styling. The model does not contain the information required to generate fine grained styling details.

While it is not always possible to uniquely assign a transformation to one of the two groups, the fuzziness introduced by the ambiguous transformation scenario are the most interesting, and the main motivation of this work. Fuzziness arises from abstraction itself. The high-level description can ignore details that are not uniquely inferable. A *Forward Engineering* approach requires such unknowns to be solved by enhancing both meta-model and transformations to remove uncertainties and maintain a unidirectional flow. This approach can lead to loss of abstraction and diminishes the benefits of the MDD methodology. User Interfaces (UI), and in particular web based ones, are a relevant example of this situation. While abstraction simplifies reasoning about the application structure and the high-level interaction, low-level details such as layout, sizing, colors or gestures can completely change the final user experience. For special domain applications a compromise can be achieved with approaches like the one implemented in commercial tools (e.g. WebRatio[3]), where the high-level model can be marked with ad hoc attributes allowing the M2T transformation to properly select a presentation template. While this approach has been successfully adopted in the industry, it introduces a layer of complexity that can easily demand more work than manual coding for application with highly specific presentation requirements. In this paper, we discuss an alternative approach, which relaxes the assumptions of *Forward Engineering* enabling the manual introduction of

[3] http://www.webratio.com/

details (e.g. styling) in the generated code, while simplifying the resolution of conflicts between the generated code and manual changes through the use of good practices in coding and development work-flows.

The paper is organized as follows: Section 2 surveys model-based and text-based approaches for the evolution of textual final products and current trends on coding practices for web applications; Section 3 presents a novel approach for the mitigation of the effects of fuzziness in M2T transformations; it uses as running example a M2T transformation generating the source code of a web application from an IFML model, while it has been implemented using the AgiLe MOdel Transformation framework (ALMOsT.js[3]) the approach is generic and can potentially be applied to every M2T transformation language; finally, Section 4 draws the conclusions and gives an outlook on future work.

2 Related work

Model to text transformations can have deep impact on an MDD work-flow. Various approaches and tools have been proposed to enhance them. Given the template-based nature of such transformations, complexity can easily arise, especially during maintenance.

Various approaches and tools have been proposed to enhance or replace M2T transformations. In [6] the automatic production of code generators from interpreters has been proposed avoiding the need for M2T transformations. In [13] a polymorphic approach has been presented, showing how modularization and dynamic invocation of templates can reduce complexity and simplify maintenance. Complexity can arise from changes in both source meta-model and target technologies. In [11] an approach to simplify M2T transformation evolution after a meta-model change is discussed. In [8] a survey of possible approaches to organize model transformations is conducted, showing the effects of moving rapidly evolving aspects of the architecture from the M2M transformations to the M2T transformations and even outside of the MDD work-flow in a manually coded abstraction layer; the study focused on SQL queries. Model and Text co-evolution has been proposed as a way to simplify M2T transformations. In [2] a bidirectional M2T transformation approach based on Triple Graph Grammar (TGG) has been proposed. The Abstract Syntax Tree (AST) representation of the target language is used in a bidirectional M2M transformation defined via TGG. The AST is structured with particular attention to supporting extra chunks of text that can be introduced during manual modifications, but are not directly managed by the transformation. In general, solving text level co-evolution with model level approaches can reduce complexity, at the cost of defining a parser specific for the target language. In [9] a trace based framework for change retainment has been proposed, which could be potentially applied in this scenario. Increasing the complexity of M2T transformations can drastically increase computation time. In [14] manual and automatically generated signatures (small, efficiently computable proxies to the final text) are exploited to increase the performance of the MDD pipeline.

Conflict Resolution at text level has been studied for a long time. Version Control Systems (VCS) like Git[4], Mercurial[5] or SVN[6] need to manage conflict resolution, in particular if working in a distributed environment. Coarse grained/language agnostic [12] or fine grained/language specific [6] automatic resolution approaches can be applied, leaving the manual intervention of the developer as a fall-back.

Separation of Concerns. In Web Applications, UI development demands a sharp division between structure (HTML) and style (CSS) for easy adaptation to various devices and clients capabilities [10]. Complexity of UI layout and styling is shifted from HTML, which describes the structure and semantics of the content, to CSS. While this scenario enables advanced use-cases, in practice it may not achieve a real separation of concerns in all cases. To obtain advanced layout and styling, the CSS rules become dependent on the HTML structure, which increases complexity, maintenance cost, and code duplication.

In the past years, coding practices shifted towards an effective compromise between separation of concerns and development costs. Modern CSS frameworks, such as Bootstrap[7], Zurb Foundation[8], Materialize CSS[9], and many others, have shown how sharing layout concerns between HTML and CSS layers can enhance re-usability and eventually reduce development time. The same trend can be seen in the field of Mobile Applications with Framework7[10], Flutter[11] and many others. This compromise blurs the line between structure and styling making more and more difficult for M2T transformations to avoid conflicts at code level.

3 An M2T based hybrid automatic and manual approach for Web application development

We propose a two steps approach to M2T transformations, which leverages both MDD and manual coding, aiming at the reduction of maintenance costs. We use as running example a code generator for web applications presented in [4], which applies forward engineering via a M2T transformation from an high-level IFML description of an application into a working prototype.

3.1 Requirements

We focus on agile development methods, such as SCRUM [15], which commands developers to create minimum viable products rapidly and evolve them via frequent iterations (sprints). As a reference usage scenario, we imagine a software team, who decides to exploit MDD in its development, by progressively introducing requirements and freely experiment with the generated code in order to enhance the user experience. Not all members of the team have a profound knowledge of the MDD pipeline and each member contributes to the project based on

[4] http://git-scm.com/
[5] http://www.mercurial-scm.org/
[6] http://subversion.apache.org/
[7] http://getbootstrap.com/
[8] http://foundation.zurb.com/
[9] http://materializecss.com/
[10] http://framework7.io/
[11] http://flutter.io/

his role and expertise. Under the above mentioned drivers, the requirements of the proposed methodology can be summarized as follows: **1) Model and Text Co-Evolution**. It must be possible to introduce new functional requirements at model level and in parallel to modify directly the code-base, e.g., to improve performance and/or styling. Changes applied to the generated code must be preserved after model change and implementation regeneration. **2) VCS Support**. The evolution of the project must be trackable using preexisting VCSs.

3.2 Intelligent Code Generation

The structure of the code generated by M2T transformations influences its maintainability. It should adhere to specific, possibly preexisting, best practices and coding styles, to facilitate code maintenance and conflict resolution. The possible aids can be divided in two main categories.

Project Structure. Modularization and separation of concerns improve software maintainability. In the web domain, approaches for code and markup modularization, such as CommonJS[12], Asynchronous Module Definition (AMD)[13] and Web Components[14], have been so successful to influence such standards as ECMAScript 6 [1] and HTML5 [16]. Splitting the generated code both logically (components, modules, . . .) and physically (files and folders) helps achieve both model and text co-evolution and VCS support: it is easier for VCSs to identify the updated files and for humans to contextualize and solve conflicts. Similarly to *protected areas* in Acceleo[15], a well structured project can automatically concentrate manual editing to specific areas of the code (e.g. GUI, service endpoints, . . .) and leave the majority of the generated code untouched (e.g., code devoted to orchestration, routing, . . .).

Coding Style. When files can be thoroughly affected by both model changes and manual editing languages, specific approaches can be applied. Modern languages give to developers freedom over many non functional aspects of the code. White-spaces and new-lines, identifiers names, order invariant statements can lead to really different appearance, which retain the same semantics. Making both developers and M2T transformations follow the same coding conventions can help to address requirement #1, reducing friction between developers and automatically generated code, as if the code generator were just another member of the team. Common VCSs, such as Git, apply text based conflict resolution algorithms [12], many of which are line based. Coding conventions can be enhanced in order to facilitate such procedures. Some examples are: splitting statements that can be possibly effected by both model changes and manual coding;

```
<!-- The class field is possibly effected by manual changes while the
     data-bind field can be changes by the generator. -->
<div class="list-item" data-bind="text: fields['title']"/>
<!-- Splitting the tag in two avoids conflicts on the same line. -->
<div class="list-item"
    data-bind="text: fields['title']"/>
```

[12] http://wiki.commonjs.org/wiki/Modules
[13] http://github.com/amdjs/amdjs-api
[14] http://www.webcomponents.org/
[15] http://www.eclipse.org/acceleo/

facilitate the insertion of new code without affecting nearby lines.

```
// Introducing a new item will mark the entire line as changed
var items = ['item1', 'item2'];
// Introducing a new items will not mark the entire array as changed
var items = [
    'item1',
    'item2', // The comma simplifies the insertion of new lines
];
```

Given the language-specific nature of code organization practices, we will not enter into further details.

3.3 Conflict Resolution Strategy

Manual editing of the generated code (requirement #1) inevitably produces conflicts, which are not different from the ones that arise from a classic VCS based distributed work-flow (requirement #2). Different developers work at the same time starting from different versions of the same code-base. Each developer needs to synchronize its local copy of the repository with the central one, solving potential conflicts that arise, before the changes are accepted. The code generator can be treated as yet another developer, who applies changes on an outdated repository. We will just consider conflicts that arise on the source code, we will ignore concurrent changes on the model. The work-flow can be organized as follow: **1)** The initial model is constructed and the code generator is run the first time. This revision of the source code G_1 (1st Generated) is sent to the central repository. **2)** A developer introduces a new manual change, starting from G_1, producing revision M_1 (1st Manual change) which is sent to the central repository. **3)** Concurrently, another developer introduces a new manual change, also starting from G_1. During synchronization possible conflicts are identified and solved producing M_2 (2nd Manual change) which is sent to the central repository. **4)** A change at model level is applied and the code generator is run again. By applying this new version on top of G_1 the generator is comparable to the second developer. During synchronization possible conflicts are identified and solved, by reapplying all the deltas introduced by each revision. Two new revision are sent to the central repository; G_2 containing the generated code and $M_{1,2}$ reintroducing the manual changes of M_1 and M_2 over G_2.

Figure 1 shows the revision history resulting from various reiterations of the proposed approach. The developer always sees a central repository, which is aligned with all the changes manually introduced. The code generator can be considered as a developer always out-of-sync who applies changes on top of the latest G_i revision. The introduction of an artificial, purely generated, M_i revision has the unwanted side effect of polluting the history with highly

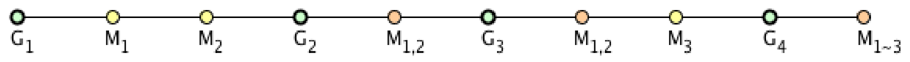

Fig. 1. Revisions history.

redundant code decreasing the benefits of a VCS system (requirement #2). The proposed approach schema can be enhanced to reduce this effect. The list of files effected by a model change is generally small compared to the whole code-base. It is possible to exploit this characteristic to reduce the impact of G_i and $M_{1\sim j}$ revisions by concentrating on the files that are actually effected. After code generation, conflicts are addressed on a per file base. Each project file can be in 4 possible states: **1)** The file was not part of the code-base (effect of a constructive change in the model). It is added in the G_i revision as a newly created file. **2)** The file was created in a G_i revision but is not part of the generated code anymore (effect of a destructive change in the model). It is removed from the code-base and the changes applied to it in any M_j or $M_{1\sim j}$ revision are discarded. **3)** The file was created in a G_i revision and it is not changed. It can be skipped and the current state of the file is preserved. **4)** The file was created in a G_i revision and it is changed. Synchronization will start from the latest G_i revision in which it is contained. All the changes applied after G_i, both in M_j or $M_{1\sim j}$ revisions, are reapplied and manual interventions is requested if automatic resolution fails. The file is added to the current G_i revision and the version after the resolution of the conflicts is added to the current $M_{1\sim j}$ revision.

4 Discussion and Future Work

This paper presented an approach for model and text co-evolution with particular attention in conflicts prevention and conflict resolution via VCS work-flows.

The approach is being used in the MDD development of web and mobile applications[16] for a energy demand management project, in which multiple editions of a energy awareness game are produced in rapid sprints, to support a user-centric design cycle in which stakeholders contribute to the design of applications. The described approach has been applied to the generation of game versions from IFML models, with a hybrid approach [4], in which both the presentation code and the code for connecting the game to a back-end cloud platform are added manually. While the described transformation architecture introduced extra conflicts resolution time after each model iteration it was compensated by a lower complexity and a shorted time to market.

The future work will focus on the experimentation and further assessment of the proposed approach in the industry, with two scenarios: companies that do not yet use MDD in their practices, to understand if introducing MDD without the disruption of existing development practices lowers the reluctance of traditional developers towards modeling; companies already applying in-house domain specific models and code generation techniques, to understand the added value of a mixed approach between MDD and manual coding.

Acknowledgments. This work has been partially supported by the European Community, through the H2020 project enCOMPASS (Grant #723059).

[16] http://play.google.com/store/apps/details?id=com.eu.funergy

References

1. ECMAScript 6. http://www.ecma-international.org/ecma-262/6.0/
2. Anjorin, A., Lauder, M.P., Schlereth, M., Schürr, A.: Support for bidirectional model-to-text transformations. ECEASST (2010)
3. Bernaschina, C.: ALMOsT.js: An agile model to model and model to text transformation framework. In: Web Engineering - 17th International Conference, ICWE (2017)
4. Bernaschina, C., Comai, S., Fraternali, P.: IFMLEdit.org: Model driven rapid prototyping of mobile apps. In: 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE (2017)
5. Bernaschina, C., Comai, S., Fraternali, P.: Formal semantics of OMGs Interaction Flow Modeling Language (IFML) for mobile and rich-client application model driven development. Journal of Systems and Software (2018)
6. Birken, K.: Building code generators for dsls using a partial evaluator for the xtend language. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA (2014)
7. Comai, S., Fraternali, P.: A semantic model for specifying data-intensive web applications using webml. In: Proceedings of SWWS'01, The first Semantic Web Working Symposium (2001)
8. García, J., Díaz, O., Cabot, J.: An adapter-based approach to co-evolve generated SQL in model-to-text transformations. In: Advanced Information Systems Engineering - 26th International Conference, CAiSE (2014)
9. Goldschmidt, T., Uhl, A.: Retainment policies - A formal framework for change retainment for trace-based model transformations. Information & Software Technology (6) (2013)
10. Hall, C.A.: Web presentation layer bootstrapping for accessibility and performance. In: Proceedings of the International Cross-Disciplinary Conference on Web Accessibility, W4A (2009)
11. Hoisl, B., Sobernig, S.: Towards benchmarking evolution support in model-to-text transformation systems. In: Proceedings of the 4th Workshop on the Analysis of Model Transformations co-located with the 18th International Conference on Model Driven Engineering Languages and Systems, MODELS (2015)
12. Horwitz, S., Prins, J., Reps, T.W.: Integrating non-interfering versions of programs. In: Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages (1988)
13. Kövesdán, G., Asztalos, M., Lengyel, L.: Polymorphic templates: A design pattern for implementing agile model-to-text transformations. In: 3rd Workshop on Extreme Modeling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems, XM@MoDELS (2014)
14. Ogunyomi, B., Rose, L.M., Kolovos, D.S.: Property access traces for source incremental model-to-text transformation. In: Modelling Foundations and Applications - 11th European Conference, ECMFA (2015)
15. Schwaber, K., Beedle, M.: Agile Software Development with Scrum. Upper Saddle River, NJ, USA, 1st edn. (2001)
16. World Wide Web Consortium (W3C): HTML5. https://www.w3.org/TR/html5/