POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione

Master of Science in Computer Science and Engineering

An approach for integrating code generation and manual development with conflict resolution

Supervisor: Prof. Piero Fraternali
Assistant Supervisor: Dr. Carlo Bernaschina

Master Graduation Thesis
Emanuele Falzone
877923

Academic Year 2017/2018

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. 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 graphical user interface and the connection between the client side and the back-end service endpoints.

Sommario

L'approccio Model Driven Development richiede strumenti adeguati per ricavare il codice di implementazione dai modelli di applicazione. Tuttavia, l'uso di generatori di codice può interferire con le pratiche di sviluppo e manutenzione del software, poiché la maggior parte degli strumenti non è in grado di conservare le modifiche manuali al codice quando il codice di implementazione viene rigenerato dai modelli. In questa tesi presentiamo un approccio che organizza le regole di trasformazione del modello e l'architettura del codice in un modo che preserva le parti del codice che sono definite al di fuori del ciclo di modellazione e generazione del codice, come il codice che definisce l'aspetto grafico dell'interfaccia utente e la connessione tra il lato client e gli endpoints del servizio back-end.

Acknowledgements

First of all I would like to thank Prof. Piero Fraternali and Dr. Carlo Bernaschina for giving me the opportunity to work with them on this interesting topic.

I would also like to thank all my friends and colleagues who have helped and contributed to this work.

The biggest thanks are for my parents (Salvatore and Gabriella), my syster (Eliana) and my grandmother (Giovanna) for the great support and motivation they have given me, and for their enormous patience.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Time to market is very important in the IT industry. Companies are constantly looking for tools that allow the developers to obtain a high quality final product as quickly as possible.

Model Driven Development has emerged as one of the leading approaches for enabling rapid, collaborative application development.

1.1 Model Driven Development

Model Driven Development (MDD) is the branch of software engineering that advocates the use of *models* and of *model transformations* as key ingredients of software development. The use of models allows developers to focus on key aspects of the system while omitting details of secondary importance. A model is an abstract representations of a system that conforms to a unique meta-model. A meta-model defines the modeling language, i.e. the constructs that can be used to express models. A model conforms to its meta-model in the way that a computer program conforms to the grammar of the programming language in which it is written.

Developers use modelling languages to specify system requirements under one or more perspectives. We can group modeling languages in two classes:

1. General purpose modeling languages, such as UML (Unified Modeling Language [1]), are intended to provide a way to visualize the design of a system.

2. Domain specific modeling languages, such as IFML (Interaction Flow Modeling Language [2]), allow the developer to represent one or more specific facets of a system.

Domain-specific modeling languages tend to support higher-level abstractions than general-purpose modeling languages, so they require less effort and fewer low-level details to specify a given system.

The other key components of Model Driven Development are model transformations. Model transformations can be thought of as programs that take models as input. Distinguishing the type of output generated by the model transformations we can classify the transformations into:

1. Model to Model (M2M) transformations.

   They translate between source and target models, which can be instances of the same or different meta-models. Figure 1.1 shows an example of Model to Model transformation where Place Chart Nets (PCN) are used to describe the semantics of IFML [3]. In [4], Statecharts are used to define the semantics of WebML, in [5] GraphQL is used to express queries on Web API's inferred from a structural (UML) and behavioral (IFML) descriptions.

2. Model to Text (M2T) transformation.

   They focus on the generation of textual artifacts from models. Figure 1.2 shows a Model to Text transformation where a GUI implementation is generated from an application model [6]. The transformation generates HTML code from the corresponding elements in the input model. Another example is described in [7] where a RESTful Web API implementation is generated from a high level description.

1.1.1 Key aspects

Abstraction is far the most key aspect of Model Driven Development. It allows developers to represent important aspects of a system, omitting details of secondary importance. However if the target meta-model has an higher expressive power than the source meta-model, the latter may not contain enough information to drive the production of a unique output, due to its abstraction level.

A relevant example of this situation are User Interfaces (UI), and in particular web based ones. Model abstraction enables reasoning about

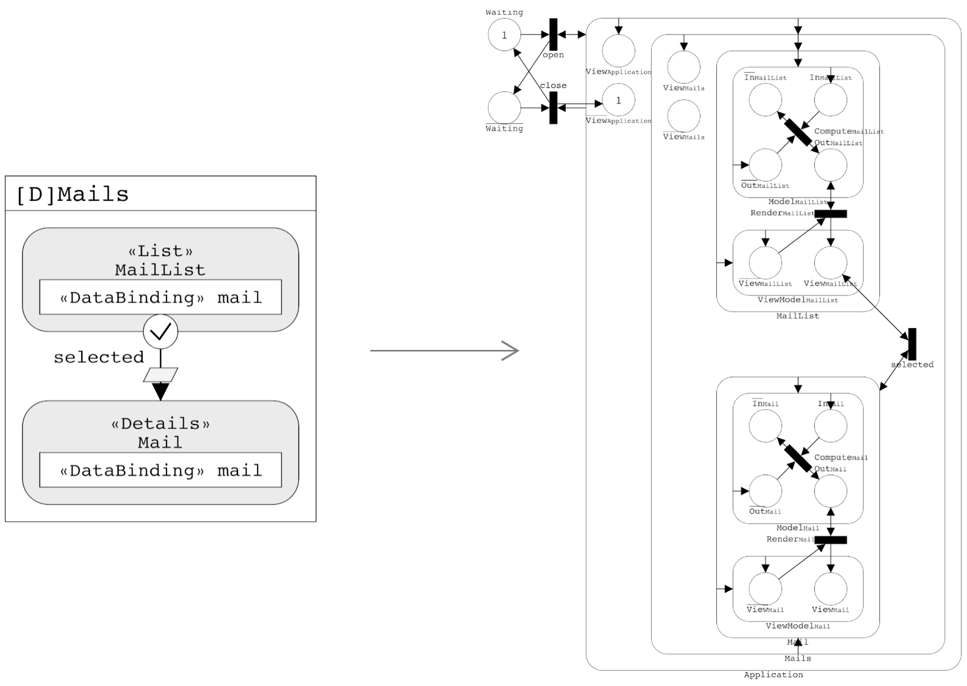
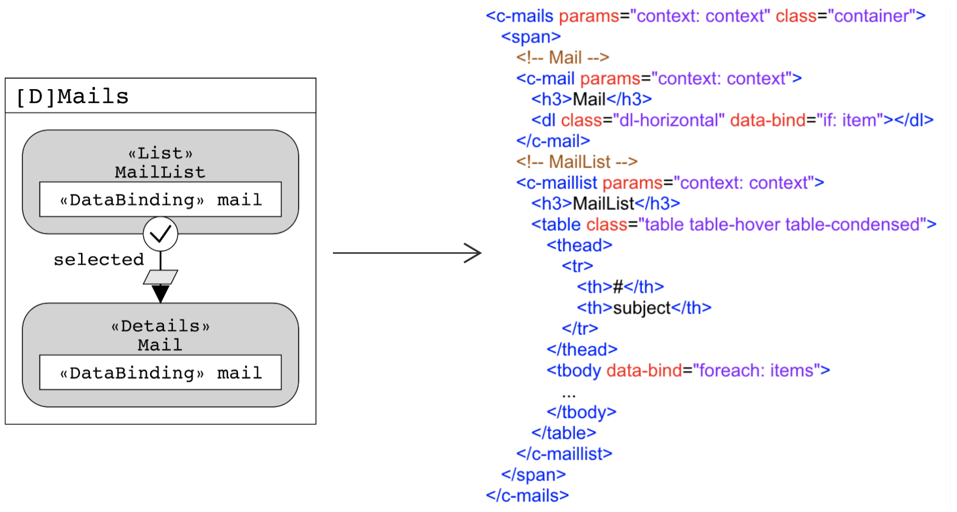


Figure 1.1: Example of Model to Model transformation



```
<c-mails params="context: context" class="container">
  <span>
    <!-- Mail -->
    <c-mail params="context: context">
      <h3>Mail</h3>
      <dl class="dl-horizontal" data-bind="if: item"></dl>
    </c-mail>
    <!-- MailList -->
    <c-maillist params="context: context">
      <h3>MailList</h3>
      <table class="table table-hover table-condensed">
        <thead>
          <tr>
            <th>#</th>
            <th>subject</th>
          </tr>
        </thead>
        <tbody data-bind="foreach: items">
          ...
        </tbody>
      </table>
    </c-maillist>
  </span>
</c-mails>
```

Figure 1.2: Example of Model to Text transformation

the organization of the front-end and about the essential interactions, regardless of presentation details. However aspects, such as content layout, fonts, colors, or gestures, must be addressed after the core GUI design is complete, because their quality greatly influences the final user experience.

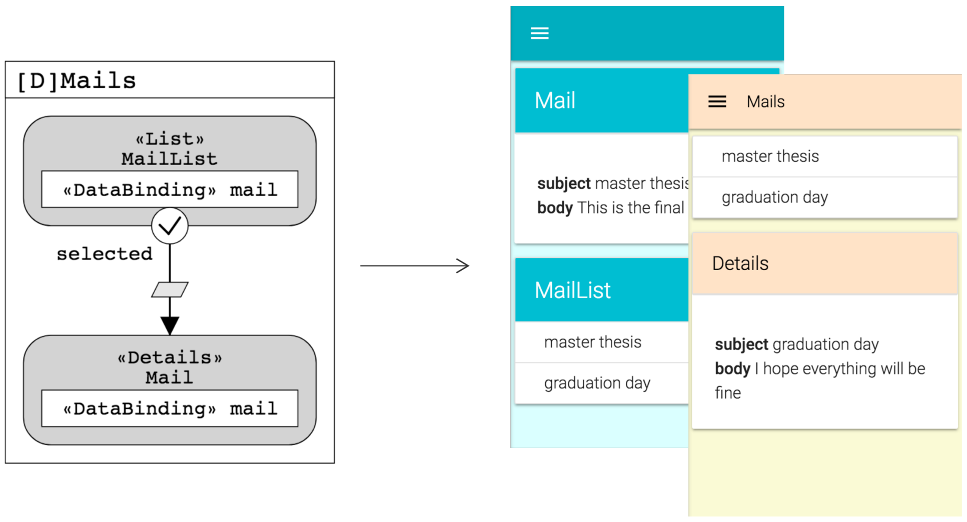


Figure 1.3: Multiple valid target from the same input model

Figure 1.3 shows an example where GUI implementation is generated from an application model [6]. Starting from the same IFML model more than one possible visual representation can be produced, by taking different decisions about the layout and the style. Another example is the generation of a RESTful Web API implementation from a high level description [7]. Alternative implementations can be produced, by taking different decisions about various aspects of the architecture such as storage and security.

Transformations should map each valid source model into one valid target model deterministically. The developer is responsible of directing the transformation so to select the alternative that best suits the requirements not captured by the input model.

Pure Forward Engineering

A pure *Forward Engineering* approach, whose flowchart is shown in Figure 1.4, requires unknown details to be solved by enhancing both the source meta-model and the transformations, to remove uncertainties and maintain a unidirectional flow from model to code.

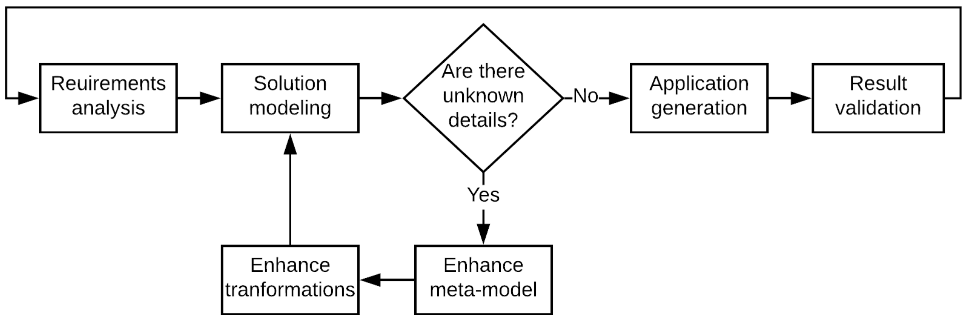


Figure 1.4: Flowchart describing Forward engineering approach

However, such an approach can lead to loss of abstraction in the source model and diminishes the benefits of the MDD methodology.

Template based forward engineering

Inspired from the forward engineering approach, template-based approach has been successfully adopted in the industry. It advocates the use of templates, i.e. code skeletons that the generator must fill-in to produce the executable code, to build Model to Text transformations. The transformation exploits the expressive power of the target meta-model to expand to a greater detail the concepts defined in the source model. The transformations select specific features of the output not inferable from the input model.

Many commercial tools such as WebRatio [8], Mendix [9], Outsystem [10] and Zoho Creator [11] require developers to annotate high-level models with ad hoc attributes, so that the M2T transformation can select the proper presentation template and create the desired output. The template-based approach introduces an additional source of complexity: the construction of templates could demand more work than manual coding of the final artifact, especially for applications with very specific presentation requirements. The construction of templates usually demands some extra knowledge: besides knowing the programming language to be generated, the developer must know the language used to build the template logic.

Problem Statement

The alternative approach to the use of template-based transformations is to exploit MDD tools for the creation of the first prototype of the application. The developers are then allowed to manually extend the code to incorporate the aspects abstracted by the input model and by the transformations that apply to it.

The obvious downside of this method is the misalignment between the model and the code. After a model change the code generated does not preserve the manual changes that the developer implemented. This problem hinders the use of transformation throughout the whole life cycle of the application.

The thesis focuses on the development of projects involving MDD tool-chains, with particular attention to the problems of co-evolving the model and the code. We propose a workflow whereby MDD tools and developers are considered as equals and can both update the source code of the application, in a way that preserves the modifications introduced by both human and automated developers.

1.2 Contributions of the thesis

The contributions of the thesis can be summarized as follows:

- We propose a distributed development and co-evolution workflow, which considers developers and MDD tool-chains as peers and helps them cooperate. The workflow captures the similarities between distributed development and model and code co-evolution and resolves conflicts between the code produced by developers and by tools in a general way, which does not depend on the specific characteristics of the model transformation tools.

- We identify the families of conflicts that arise in a workflow integrating human coders and automated tools, characterize their origins and propose both automatic resolutions or mitigation strategies.

- We showcase the power of the proposed approach via a reference implementation and evaluate its impact on two use-case applications.

1.3 Document Structure

The thesis is organized as follows: Chapter 2 will survey model-based and text-based approaches for the evolution of textual artifacts. Chapter 3 will propose a workflow which allows developers and MDD tool-chains to co-evolve the same code-base and survey the type of conflicts that can arise and how to prevent or mitigate them. Chapter 4 will discuss the algorithm from a practical point of view presenting a reference implementation. Chapter 5 will evaluate the effect of the proposed approach on the development of two projects. Finally, Chapter 6 draws the conclusions and gives an outlook on future work.

Chapter 2

Related Work and State of the Art

In this Chapter we will survey various techniques that allow developers to build customized applications using Model Driven Development tools. According to the *forward engineering* approach developers have to enhance both the meta-model and the transformations. In this way developers are able to specify the details that are not inferable from the original model. In Section 2.1.1 we will survey the techniques used to enhance the meta-model. In Section 2.1.2 we will survey the techniques used to enhance or replace model transformations, with particular focus on model to text transformation and template based approach. In Section 2.2 we will survey model and code co-evolution techniques that allow the developers to synchronize models and source code. At the end, in Section 2.3 we will survey state of the art tools that allow developers to work concurrently.

2.1 Forward engineering approach

The source meta-model, due to its abstraction level, may omit details needed to drive the model transformation to produce a unique output. A *Forward Engineering* approach requires such details to be specified enhancing both the source meta-model and the model transformations.

2.1.1 Evolution of meta-models

Although modeling is an activity regulated by meta-models, currently there are no commonly accepted mechanisms to define how meta-models can be extended. In [12] the authors propose a mechanism that allows specifying customization and extension rules for meta-models. The proposed approach has the advantage that it is non intrusive, and generic, that is, extension rules can be linked to any meta-model.

In [13] the authors focus on improving the agility of modeling frameworks by allowing them to be more flexible and adaptable to changes on the meta-models they provide support for. They propose a lightweight meta-model extension mechanism, based on a textual domain specific language for specifying meta-model extensions.

In [14] the authors use UML as reference modeling language. It has been widely used for modeling applications and changes continuously. They analyze the evolution of the meta-model by using complex network and information entropy technologies. The approach can provide insight into the constructive mechanism and future trends of the examined meta-model.

Meta-models are evolving over time, requiring existing domain models to be co-evolved. Various solutions have been proposed to solve the problem of meta-model and model co-evolution. A vision of co-evolution between meta-models and models through consistent change propagation is presented in [15, 16]. The approach addresses co-evolution issues without being limited to specific meta-models or evolution scenarios. It relies on incremental management of meta-model-based constraints that are used to detect co-evolution failures, generating suggestions for correction when a failure is detected.

The high number of solutions makes it difficult for practitioners to choose an appropriate approach. In [17] a survey of approaches to support meta-model and model co-evolution is presented, introducing a taxonomy of solution techniques and classifying the existing approaches. They also use the results to provide a decision support for practitioners, who aim to adopt solutions from research.

2.1.2 Evolution of Model-to-Text transformations

Model to text transformations are at the base of MDD workflows. Given the template-based nature of most such transformations, complexity can easily arise from the creation and maintenance of templates [18]. Various approaches and tools have been devised to enhance or replace M2T transformations.

In order to avoid the need of M2T transformations, in [19] the authors show how efficient, platform-specific code can be generated out of DSL programs using specialization of an interpreter for the DSL by partial evaluation.

In [20] the authors propose a design pattern that allows for the decomposition of complex templates with branching and conditions inside into simpler ones. The code generator does not know about the concrete templates that are called. The pattern results in a flexible code generator with simple templates, good extensibility and separation of concerns. This agility also facilitates the design for extension and changes.

Complexity can arise from changes in both source meta-model and target technologies. In [21] the authors propose a definition for a standard problem to evaluate the evolution support in M2T transformation systems, with the objective to allow for benchmarking of multiple evolution-support techniques for M2T transformations.

In [18] 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 workflow as an abstraction layer managed manually.

The above mentioned approaches exploit specific features of the input and output meta-models and of the transformation infrastructure. The approach discussed in this thesis moves a step further, overcoming dependencies from languages and use-cases. Instead of defining a strong separation between the code generated by M2T transformations and the code managed manually, we investigate a general-purpose method, independent of the input and output meta-models and of the transformation tools, in which the boundaries between the two regions of the code (human and tool-generated) can be set flexibly and changed over time.

2.2 Model and code co-evolution

Model and code co-evolution techniques have been explored to simplify M2T transformations. In [22] a bidirectional M2T transformation approach based on Triple Graph Grammar (TGG) is discussed. 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, addressing text-level changes by lifting them at the model level can reduce the complexity of modeling and transformation, at the cost of defining a parser specific for the target language and a reverse mapping from low level code to the high level concepts, which is normally defined for a limited class of code-level patterns. In [23] a trace based framework for change retainment is proposed, which helps tracing back code-level modifications to the model. Both bidirectional and trace-based approaches are bound to the target languages.

The need to support different languages or new features of a target language requires changes to be propagated to the transformation and tracking chains. The approach proposed in this thesis poses minimal requirements on the way M2T transformations should structure the generated code and works mostly on the output of the transformations, in a language and tool-independent way.

Approaches have been proposed to allow the developers to manually modify the output of Model to Text transformations while preserving such manual modification across multiple execution of the transformations. Acceleo [24] use *protected areas* to automatically concentrate manual editing to specific portions of the code and leave the majority of the generated code untouched. However a sharp division between structure and style is not always achievable, especially when rapid evolution is required. Many modern frameworks, such as Framework7 [25], Flutter [26] and others moved towards an effective compromise between separation of concerns and development costs.

2.3 Distributed development

Distributed development is essential to speed up application development. Developers need tools that allow them to cooperate easily and to provides control over changes to source code. Version Control Systems (VCS) have been developed to fulfill this requirement.

Version Control Systems like Git [27], Mercurial [28] or SVN [29] also manage conflict resolution.

Conflict resolution at text level has been studied for a long time, providing automatic resolution and leaving the manual intervention of the developer as a fall-back. In [30] a coarse grained technique is presented to provide the foundation for building an automatic program-integration tool. In [31] the proposed algorithm exploits the fine-grained edit operation history of Java source code and extracts only the edit operations that affect the revision of a particular class member alleviating the task of merging conflicting revisions.

Our approach exploits the well-proved capabilities of conflict resolution tools to support the co-evolution of the application model and of code generated from it, by enabling the merge of the code produced by programmers and by model transformations. Even if not all conflicts can be resolved automatically, in Chapter 5 we show that the number of conflicts that require human intervention is comparable or less than the number of changes at transformation level needed to obtain the same result.

Chapter 3

Proposed Approach

In Section 3.1 we analyze how the software developers' community deals with parallel development of software features. Then, in Section 3.2, we show similarities and differences between parallel development and model and text co-evolution. In Section 3.3 we propose a solution to apply the parallel development methodologies to the co-evolution of model and text, by treating the MDD tool-chain as yet another developer in the team. Finally, in Section 3.4 we investigate on how the proposed approach can be integrated with automatic conflict resolution.

Given the generic nature of the approach, the described work-flow is independent of the Version Control System (VCS), modeling and development language, and transformation framework.

Before proceeding, we introduce the concepts and notations used in the rest of the thesis.

- **Developer**: D_i denotes the i-th member of the development team.
- **Local/central code base**: C_i denotes the version of the code-base edited by developer D_i. C_C identifies the central code-base shared among developers.
- **Revision**: $R_{i,j}$ denotes the j-th revision of code-base C_i; it is the full textual artifact stored in C_i at a particular point in time. $R_{C,j}$ denotes a revision in the central code base.
- **Equivalence**: two revisions $R_{i,j}$ and $R_{m,n}$ are equivalent, if the content of the textual artifact stored in them is the same.

- **Difference**: the difference $R_{i,j} - R_{m,n}$ of two revisions is the set of changes that need to be applied to $R_{m,n}$ to produce $R_{i,j}$.

- **Delta**: the delta introduced by $R_{i,j}$ ($\Delta_{i,j} = R_{i,j} - R_{i,j-1}$) is the difference between $R_{i,j}$ and the previous revision $R_{i,j-1}$.

- **Conflict**: let n be the number of revisions in the central code-base C_C; a revision $R_{i,j}$ is said to be in conflict with C_C if $R_{i,j-1}$ in not equivalent to $R_{C,n}$, or if j is equal to 1. In other terms, a conflict signals that a new revision has been produced starting from a newly initialized code-base or from a version different from the last one consolidated in the central code base.

- **Submission**: let n be the number of revisions in the central code-base C_C; the submission $R_{i,j} \rightarrow C_C$ is the act of creating a shared revision $R_{C,n+1}$ in C_C which is equivalent to the local revision $R_{i,j}$. A submission is always performed from a local code-base C_i to the central code-base C_C. If $R_{i,j}$ is not in conflict with C_C, the submission $R_{i,j} \rightarrow C_C$ generates $R_{C,n+1}$, in such a way that $\Delta_{C,n+1}$ is equivalent to $\Delta_{i,j}$. Note that $R_{i,j}$ is equivalent to $R_{C,n+1}$, by the definition of submission, and $R_{i,j-1}$ is equivalent to $R_{C,n}$, by the definition of conflict. The submission of a non-conflicting revision $R_{i,j}$ is always accepted. The submission of a revision in conflict is by default **rejected**, to avoid the possible loss of local changes and prompt the developer to solve the conflict. A conflicting submission can be **forced**, overriding the default.

- **Conflict Resolution**: let n and m be the number of revisions in the local code-base C_i and in the shared code-base C_C respectively; the conflict resolution $R_{C,m} \mapsto C_i$ is the act of creating a (local) revision $R_{i,n+1}$ based on both $R_{C,m}$ and $R_{i,n}$. Note that $R_{i,n+1}$ may be equivalent neither to $R_{C,m}$ nor to $R_{i,n}$. The objective of conflict resolution is to enable a submission from a developer even if he has worked on a version that is out of synch with respect to the central code base. Solving a conflict between $R_{C,m}$ and C_i allows a submission $R_{i,n+1} \rightarrow C_C$ to be performed, even if $R_{i,n} \rightarrow C_C$ was previously rejected due to a conflict.

  When a revision $R_{i,n}$ is in conflict with C_C the developer can solve the conflict by generating a new revision $R_{i,n+1}$. Performing the operation on C_i instead of C_C allows other developers to continue their work without interference. After the resolution is performed, the developer can submit $R_{i,n+1}$ to C_C.

- We refer to a revision of the code-base that results from a manual change as $R^M_{i,j}$ (**manual revision**), to a revision containing generated artifacts as $R^G_{i,j}$ (**generated revision**), and to a revision that results from conflict resolution as $R^R_{i,j}$ (**resolution revision**).

3.1 Parallel & Distributed Development

During day to day operations, teams must deal with the problem of parallel development, whereby different subjects introduce distinct new features into the code-base independently. A common development work-flow on an ongoing version of a system is the following:

1. Developer D_1 starts the current development sprint with an empty local code base C_1.
2. She aligns to the current status of the project by initializing her local copy C_1 to contain n revisions imported from the central code base ($R_{1,j} = R_{C,j}$) for $j \in \{1..n\}$.
3. She introduces a new feature by applying changes on top of $R_{1,n}$, creating a new revision $R^M_{1,n+1}$.
4. She makes a submission to the centralized code-base generating a new revision: $R^M_{1,n+1} \rightarrow C_C = R^M_{C,n+1}$. $R^M_{C,n+1}$ is accepted, because $R_{C,n} = R_{1,n}$.
5. She deletes her local copy of the code-base C_1 returning to the initial state. Her work is safely stored in C_C.

After such a work-flow, the revision history shown in Figure 3.1 is achieved.

When two developers D_1 and D_2 work in parallel on the same system, the work-flow may look like the following:

1. Developers D_1 and D_2 create their local copy C_1 and C_2 of the code-base from the current content of the central code base C_C, which comprises n revisions; then $\forall_{j\in\{1..n\}} R_{1,j} = R_{C,j} = R_{2,j}$.

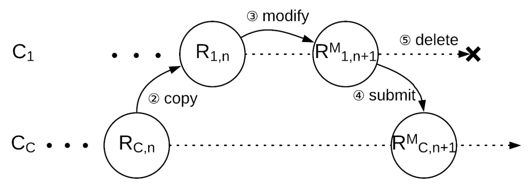


Figure 3.1: Development without Conflicts

2. D_1 introduces a new feature by applying changes to $R_{1,n}$, creating a new revision $R^M_{1,n+1}$.

3. At the same time and independently D_2 introduces another feature, by applying changes on top of $R_{2,n}$, creating a new revision $R^M_{2,n+1}$.

4. D_1 submits $R^M_{1,n+1}$ to the centralized code-base generating a new revision $R^M_{1,n+1} \rightarrow C_C = R^M_{C,n+1}$. $R^M_{C,n+1}$ is accepted because $R_{C,n} = R_{1,n}$.

5. D_1 deletes her local copy of the code-base C_1 returning to the initial state. Her work is safely stored in C_C.

6. D_2 tries to submit $R^M_{2,n+1}$ to the centralized code-base. The operation is rejected because $R^M_{2,n+1}$ creates a conflict with C_C.

7. D_2 solves the conflict between $R^M_{2,n+1}$ and C_C generating $R^M_{C,n+1} \mapsto C_2 = R^R_{2,n+2}$.

8. D_2 submits $R^R_{2,n+2}$ to the centralized code-base generating a new revision $R^R_{2,n+2} \rightarrow C_C = R^R_{C,n+2}$. The submission now is accepted because $R^R_{2,n+2}$ is the result of the conflict resolution $R_{C,n+1} \mapsto C_2$.

9. D_2 deletes her local copy of the code-base C_1 returning to the initial state. Her work is safely stored in C_C.

After such a work-flow, the revision history shown in Figure 3.2 is achieved.

This approach can scale to large number of developers who, before submitting their revision to the central code-base, are responsible to ensure that conflicts are resolved.

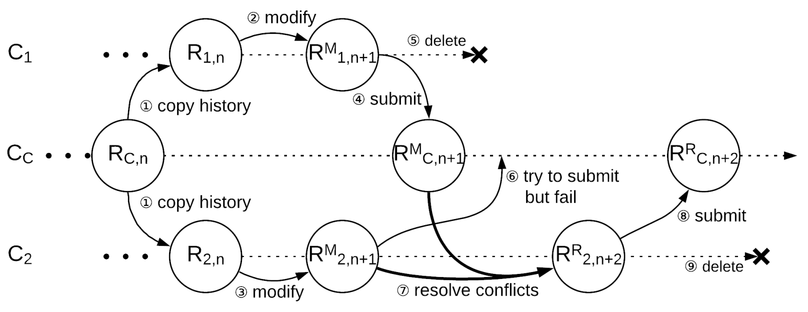


Figure 3.2: Development with Conflicts

Parallel development work-flows are commonly supported by Version Control Systems (VCSs), either distributed (e.g. GIT) or centralized (e.g. SVN). Their functionality is not limited to content transfer and history tracking. Given a local and the central code-base C_i and C_C with their respective latest revisions $R_{i,n}$ and $R_{C,m}$ VCSs can identify conflicts automatically. With the help of primitives specific to each VCS, it is even possible to automate, or at least simplify, conflict resolution. Various approaches can be applied to assist a resolution $R_{C,m} \mapsto C_i$: from simple analysis of the difference $R_{i,n} - R_{C,m}$, to the more complex analysis of the full revisions histories of both the repositories. If two equivalent revisions $R_{C,k}$ and $R_{i,k'}$ are identified in the histories of C_i and C_C, the set $\{\Delta_{C,j} \mid j \in \{k..m\}\}$ of all the deltas in that history branch can be exploited in order to be reapplied step by step. In Chapter 5 we evaluate the results of the specific technique tested in our experiments.

3.2 Model and Text Co-Evolution

In an MDD-based work-flow, model-to-text transformations produce textual artifacts (e.g. code, configuration files, documentation, ...) from models. By allowing manual editing on the generated output to specify details not covered by the input model and by the code generation rules, it is possible to introduce conflicts with subsequent transformation executions based on evolved versions of the model. A naïve approach to the resolution of such conflicts is the one realized by the following work-flow:

1. Development starts with a phase of modeling and code generation. Thus, the central code-base initially contains a single revision $R^G_{C,1}$ which is the result of the first execution of the transformation.

2. Developer D_1 creates a local copy C_1 of the code-base C_C, which contains one revision. $R^G_{1,1} = R^G_{C,1}$

3. She introduces a manual change on top of $R^G_{1,1}$ producing a new revision $R^M_{1,2}$;

4. She submits the new revision to the centralized code-base generating a new revision $R^M_{1,2} \rightarrow C_C = R^M_{C,2}$. The submission is accepted because $R_{C,1} = R_{1,1}$.

5. She deletes her local copy of the code-base C_1 returning to the initial state. Her work is safely stored in C_C.

6. She evolves the original model and is ready to execute the transformation again.

7. She executes the transformation and stores the result into an a new empty code-base C_1 initializing it with the new purely generated $R^G_{1,1}$ (different from $R^G_{C,1}$), which reflects the current state of the model at the code level. Note that manual modifications of the code introduced at step 3 are not reflected in the newly initialized local code-base, because the code generator is blind to manual modifications and would overwrite them anyway.

8. She tries to submit $R^G_{1,1}$ to the centralized code-base. The operation is rejected because $R^G_{1,1}$ is in conflict with C_C. Rejection occurs due to the way code generation works: C_1 contains just one revision and is not created by evolving a preceding shared revision in the central code base.

9. She solves the conflict between $R^G_{1,1}$ and C_C generating an updated local copy that reconciles the manual changes stored in the central code-base and the new code generated after the model update ($R^M_{C,2} \mapsto C_1 = R^R_{1,2}$).

10. She submits the new revision to the centralized code-base generating a new revision $R^R_{1,2} \rightarrow C_C = R^R_{C,3}$. The submission is accepted because $R^R_{1,2}$ is the result of the conflict resolution $R^M_{C,2} \mapsto C_1$.

11. She deletes her local copy of the code-base C_1 returning to the initial state. Her work is safely stored in C_C.

After such a work-flow, the revision history shown in Figure 3.3 is achieved.

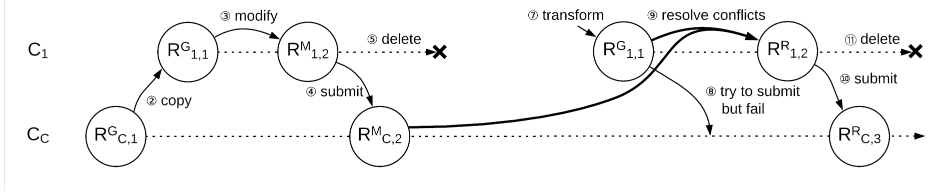


Figure 3.3: Conflict resolution after manual change

3.3 The Virtual Developer

The developer D_2 in Section 3.1 works on an outdated version of the code-base (Figure 3.2), which requires the conflict to be solved. The Model-to-Text transformation in Section 3.2 generates a revision $R^G_{1,1}$ (Figure 3.3) in conflict with the central code base due to the previous manual update of the generated code. These two scenarios are different with respect to the agent whose changes generate the conflict, but the evolution of the textual artifact follows a similar path.

In work-flow of Figure 3.3 the newly generated revision $R^G_{1,1}$ is used to initialize a new code-base C_1 due to the fact that the Model-to-Text transformation does not proceed by evolving the code-base incrementally, as real developers do, but always regenerates the whole textual artifact from scratch. At every transformation step, it would be as if an update on the entire artifact is checked in to the repository regardless of the past history. Given this observation, the work-flow of Figure 3.3 is equivalent to the one of Figure 3.4: at each model update and code generation step, the developer creates a fresh local copy C_1 of the code-base C_C, up to the latest purely generated $R^G_{1,1} = R^G_{C,1}$, before overwriting it with the new purely generated revision $R^G_{1,2}$.

It is important to notice that the initialization of the local code base with the latest purely generated revision before a code generation step makes it is possible to compute the delta $\Delta^G_{1,2}$ introduced by $R^G_{1,2}$. Even though the Model-to-Text transformation overwrites the entire artifact, $\Delta^G_{1,2}$ singles out exactly the *incremental* changes that

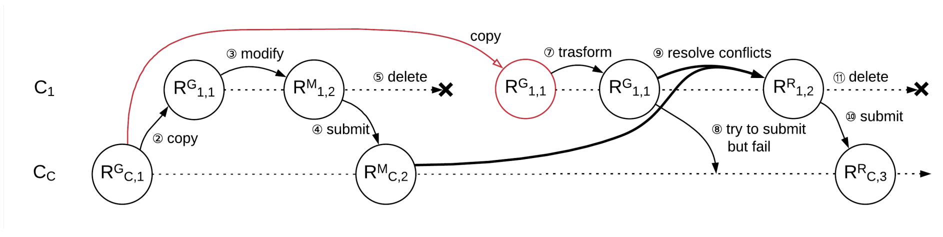


Figure 3.4: Conflict resolution after manual change, alternative path

the Model-to-Text transformation induces over the previously generated revision $R^G_{1,1}$ due to the model update. This initialization method makes the behavior of the code generator more similar to that of the human developer, who extends the code base incrementally.

Besides the difference in the initialization of the code base, the manual and automatic generation of the artifacts follow the same path, as visible by comparing the work-flow of Figure 3.2 and the one of Figure 3.4. Revisions $R^M_{2,n+1}$, of Figure 3.2, and $R^G_{1,2}$, of Figure 3.4, are comparable: both generate a delta ($\Delta^M_{2,n+1}$ and $\Delta^G_{1,2}$ respectively), which embodies the changes w.r.t. a preceding revision ($R^M_{2,n}$ and $R^G_{1,1}$) in an outdated code-base (C_2 and C_1 respectively) generating a conflict with C_C.

Given such similarity, a Model-to-Text transformation can be considered as a *Virtual Developer*, who always generates a conflict with C_C. Treating the Model-to-Text transformation as an additional developer potentially simplifies the management of manual updates in the forward engineering MDD life cycle. Tools and methodologies that are normally applied for conflict resolution among developers can be applied to both *Human* and *Virtual* Developers. Multiple *Human* developers and a *Virtual* developer can work in parallel, in a work-flow such as the following:

1. C_C contains n revisions, the latest revision $R^G_{C,n}$ is purely generated.

2. A *Human* developer D_H creates a local copy C_H of the code-base C_C. $\forall_{j\in\{1..n\}}(R_{H,j} = R_{C,j})$

3. D_H introduces a new feature by applying changes to $R^G_{H,n}$, creating a new revision $R^M_{H,n+1}$.

4. D_H submits the new revision to the centralized code-base gener-

ating a new revision $R^M_{H,n+1} \rightarrow C_C = R^M_{C,n+1}$. The submission is accepted because $R_{C,n} = R_{H,n}$.

5. D_H deletes her local copy of the code-base C_H returning to the initial state. Her work is safely stored in C_C.

6. The model is updated and the transformation is ready to be executed.

7. The *Virtual* developer D_V creates a local copy C_V of the code-base C_C *up to the latest purely generated revision*: $\forall_{j\in\{1..k\}}(R_{V,j} = R_{C,j})$, where k is the index of the last *generated* revision ($k = n$ in the current example). In this way, the Virtual Developer aligns its local code base to the status that reflects the previous version of the model.

8. D_V executes the transformation and stores the result into C_V generating a new revision $R^G_{V,n+1}$, which is a replacement of the entire textual artifact.

9. D_V tries to submit $R^G_{V,n+1}$ to the centralized code-base. The operation is rejected because $R^G_{V,n}$ is not equivalent to $R^M_{C,n+1}$, due to the intervening manual update of D_H.

10. D_V solves the conflict between $R^M_{C,n+1}$ and C_V generating $R^M_{C,n+1} \mapsto C_V = R^R_{V,n+2}$. In this step, the $\Delta^G_{V,n+1}$ is used to identify the modifications introduced by the latest round of code generation, which simplifies, and even automates in some cases, the identification and resolution of collisions with the manual modifications of the code, as explained in Section 3.4

11. In order to safely store the latest purely generated revision in the central C_C for future alignment, D_V forces the submission of $R^G_{V,n+1}$ to the centralized code-base, generating $R^G_{C,n+2}$.

12. D_V submits the $R^R_{V,n+2}$ to the centralized code-base, generating a new revision $R^R_{V,n+2} \rightarrow C_C = R^R_{C,n+3}$. The submission is accepted because $R^G_{C,n+2}$ is equivalent to $R^G_{V,n+1}$. It is important to notice that $\Delta_{C,n+3}$ is identical to $\Delta_{V,n+2}$, due to the previous forced submission, which saved in the central code base also the latest purely generated revision.

13. D_V deletes its local copy of the code-base C_V returning to the initial state. Its work is safely stored in C_C.

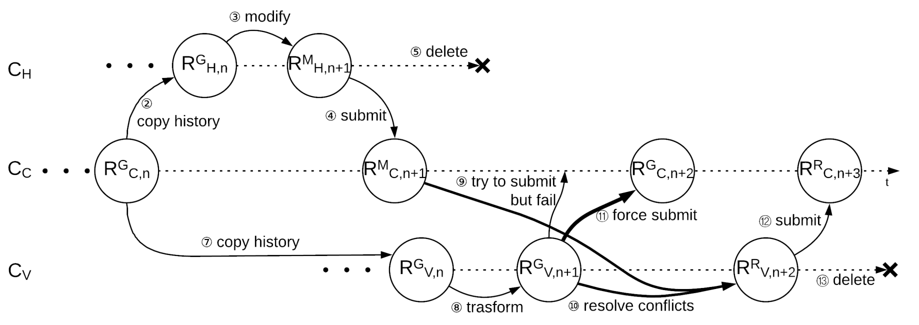


Figure 3.5: Revisions history.

Figure 3.5 shows the revision history that results from considering the Model-to-Text transformation as an additional virtual developer. The central code-base now contains a twofold sequence of revisions: automatically generated ($R^G_{C,i}$) and conflict-resolved ($R^R_{C,i+1}$). The *Human* developer always updates the latest revision, whereas the virtual developer is always out-of-sync and applies changes on the latest *generated* revision $R^G_{C,i}$.

By preserving the generated revisions ($R^G_{C,i}$) in the code-base history we ensure that the delta $\Delta^G_{V,n+1}$ can be interpreted as a mutation of revision $R^G_{V,n}$, which was generated by the previous execution of the Model-To-Text transformation. In Section 3.4 we will see how this interpretation of $\Delta^G_{V,n+1}$is key in automating conflict resolution.

3.4 Automating conflict resolution

Given the definition of *Conflict* introduced at the beginning of the Chapter we can describe the *Conflict Resolution* $R^M_{C,n+1} \mapsto C_V = R^R_{V,n+2}$ as any automated, manual or hybrid procedure which given the history of the code-base C_c up to $R^M_{C,n+1}$ and of C_i up to the latest revision $R^G_{V,n+1}$ is able to generate a new revision $R^R_{V,n+2}$.

Policies are needed to prevent the insurgence of conflicts or else to mitigate them or at least support their automatic resolution. Current MDD tools automate the conflict resolution phase in different ways.

Template-based forward engineering

They avoid conflicts a priori, by disallowing manual updates to the generated code. The revision produced by the conflict resolution $R^M_{C,n+1} \mapsto C_V = R^R_{V,n+2}$ is always equivalent to $R^R_{V,n+2}$. This is possible thanks to the assumption that the code generator is the source of truth, so any manual changes introduced on code-base C_C are ignored. This is the approach of *template-based* model-to-text transformations. The price to pay is the need of creating templates, a task that requires a kind of meta-programming, i.e., the programming of partial examples and code skeletons that the generator must fill-in to produce the executable code. This effort requires non-standard programming skills and is tied to the specificities of the MDD tool.

Protected areas

They automate conflict resolution by means of *protected areas*, i.e., identifying the regions of the code that the generator cannot overwrite. Each area is associated with a unique signature. The content of revision $R^G_{V,n+1}$ is used as base artifact to build $R^R_{V,n+2}$. If a signature is present both in $R^M_{C,n+1}$ and $R^G_{V,n+1}$ the content of the corresponding protected area is extracted from the artifact in central revision $R^M_{C,n+1}$ and inserted inside the matching protected area of the local revision $R^R_{V,n+2}$. If a signature is present in $R^M_{C,n+1}$ but not in $R^G_{V,n+1}$ the content of the corresponding protected area in the central $R^M_{C,n+1}$ is just discarded. It is important to notice that, while the developer is not involved in the conflict resolution, her intervention is needed to complete the protected areas whose signature is present in $R^G_{V,n+1}$ but not in $R^M_{C,n+1}$. This approach assumes that the aspects where human intervention is required can be sharply separated from the rest of the code, an assumption that is hardly verified in modern web and mobile applications, where the presentation is still entangled with the structure of the front-end.

Our aim is to relax these assumptions, acknowledging conflicts as the normal outcome of the integration between the virtual and human developers and performing conflict resolution following an hybrid approach which exploits the history of the repository to automatically preserve manually introduced changes and involves the developer in those few cases in which the proper resolution is not clear to the tools itself.

3.4.1 Exploiting deltas

At any given point in the history of a project, the shared code-base C_C and the local code base of the Virtual Developer C_V may have diverged, due to the introduction of updates by human developers; however, thanks to the way in which the work-flow is managed, the two code-bases have an equivalent history up to the revisions $R^G_{C,n}$ and $R^G_{V,n}$ (Figure 3.5). These revisions contain the *latest shared* output of the model-to-text transformation.

After such "synchronization point", the centrally shared delta $\Delta^M_{C,n+1}$ contains all the changes introduced by human developers posterior to $R^G_{C,n}$ and the local delta $\Delta^G_{V,n+1}$ contains all the changes introduced by the Virtual Developer in the last (yet not shared) execution of the model-to-text transformation. These two deltas may interfere and the interference must be identified and resolved.

To support the resolution of a conflict, each delta must be decomposed into the individual changes it contains. The following definitions characterize the content of a delta:

- **Line**: a tuple consisting of a unique identifier, a content and a position.
- **Artifact**: an ordered set of lines.
- **Atomic update**: an elementary modification of the artifact. Allowed atomic updates are:
    1. Insertion: creation of a new line with given content at a position successive to an existing line or at the beginning of the artifact. The positions of the lines located after the new line are incremented.
    2. Deletion: the removal of a line, with the consequent decrement of all the lines positioned after it.
- **Update**: a set of atomic updates affecting distinct lines at consecutive positions.
- **Collision**: two updates u_i and u_j are in collision if there exist an atomic update $a_i \in u_i$ and an atomic update $a_j \in u_j$ affecting the same line.
- **Update Graph**: a undirected graph U in which vertexes are updates and edges, if present, denote the collision between them.

- **Collision Group**: a collision group U_C is a connected component of the update graph having size greater than one, i.e., comprising at least one collision. An update graph with no collision groups is called a **Collision-free graph** U_F; the updates contained in a collision-free graph can be applied independently.

- **Collision Resolution**: a procedure that takes in input a collision graph and produces in output a new collision free graph.

Conflict resolution can be achieved with the following steps:

1. $\Delta^M_{C,n+1}$ is decomposed into its collision-free updates U^M_F.

2. $\Delta^G_{V,n+1}$ is decomposed into its collision-free updates U^G_F.

3. The set $U_C = U^M_F \cup U^G_F$ is formed, which can contain collisions.

4. If U_C contains no collision groups, the set of updates to apply U is defined as: $U = U_C$. Otherwise, collision resolution is applied, so to create a new collision-free graph U_F and the set of updates to apply U is defined as: $U = U_F$.

5. The updates in U, which are guaranteed non to collide, are applied to $R^G_{V,n+1}$ generating $R^R_{V,n+2}$.

The above mentioned steps are supported in all industry standard VCSs. They automatically identify the set U, by decomposing $\Delta^M_{C,n+1}$ and $\Delta^G_{V,n+1}$. If no collision groups are identified they automatically produce $R^R_{V,n+2}$. If one or more collision groups are identified they support the developer during collision resolution.

3.4.2 Reducing the number of collisions

Supposing that all the changes manually introduced by the human developer in the generated code are needed (thus the size of $\Delta^M_{C,n+1}$ cannot be reduced) the number of collisions can be reduced in tow ways:

1. Making human developer and virtual developer work on different lines.

2. Reducing the size of $\Delta^G_{V,n+1}$.

Separation of concerns

The ability to identify areas (or even entire files) which are pure responsibility of the model-to-text transformation or, viceversa, are purely responsibility of the human developer can simplify conflict resolution. While perfect separation of concerns may not be always reachable, due to limitations of the target language or because the complexity which needs to be added to the project is not justifiable, it is always advisable.

Files which should not be changed by the developer will never generate conflicts, due to the fact that the only delta which introduces changes are $\Delta_{V,n+1}$.

Files which should be edited by the developer should be generated following a fixed and recognizable template, in order to make only $\Delta_{M,n+1}$ the one which introduces changes. Given the fact that these files are introduced and removed in $\Delta_{V,n+1}$ revisions and editing only in $\Delta_{M,n+1}$ revisions the resolution can be naïve.

Separation of concerns can be applied even at the level of groups of lines or even single lines of text. Lines which are responsibility of both the developer and the transformation should be split exploiting one of the language invariabilities.

For example line level separation of concerns can be achieved by splitting HTML tags from a single line

```
<div class="list-item" data-bind="text: fields['title']"/>
```

to multiple lines containing each one a different attribute. Attributes related to styling, responsibility of the developer, can be separated from the ones related to behavior, responsibility of the model-to-text transformation, avoiding collisions.

```
<div class="list-item"
     data-bind="text: fields['title']"/>
```

Accurate design of the transformations

Reducing the number of lines modified by the code generator decreases the likelihood of collisions.

A transformation should always deterministically produce the same code from the same model, also it should produce the same code from equivalent models, i.e. models differing just on meta-data (e.g. graphical information used during rendition in an editor). Non determinism typically arises from the non-deterministic iteration over model elements during code generation. It can be avoided if the transformation exploits a fixed criterion for model navigation based on semantically meaningful features, even if such criterion is not part of the modeling language. For example, a model-to-text transformation generating code from an IFML DataFlow element can generate different artifacts depending on the order in which the parameter bindings are traversed. The following JavaScript code could be produced from an IFML data flow associated with two parameter bindings (title and author of a song).

```
var packet = {
   'title' : data['song'],
   'author' : data['author_name']
};
```

If the order in which the parameter binding sub-elements are traversed is not deterministic, the transformation can produce different outputs between two runs on the same input model.

```
var packet = {
   'author' : data['author_name'],
   'title' : data['song']
};
```

Chapter 4

Implementation Details

In this Chapter we will analyze how the workflow proposed in Section 3.3 can be mapped to a Git [27] workflow and present a reference implementation that we have used during our tests.

The proposed approach can be implemented with various Version Control System. For each VCS a specific workflow is needed. Git [27] as a distributed VCS, is aimed at speed, data integrity, and support for distributed workflows.

4.1 Git primitives

The concepts introduced in Chapter 3 can be mapped to concrete Git primitives as follows:

- *code-base*s are mapped to Git *branches*, i.e., parallel histories inside a single Git repository.
- *revision*s are mapped to Git *commit*s. Each *commit* is associated with a message and a *hash* computed as SHA-1 checksum.
- the act of copying the central code-base is mapped to the *clone* or *branch* Git operations, depending on the location of the central code-base; the former in the case of a centralized repository, the latter in the case of a local branch.
- *submission* is mapped to the Git *push* operation, which copies commits from the current branch to another local or remote one. The *push* operation may fail if the latest commit in the remote

branch (i.e., the *HEAD* of the branch) is not identified in the local branch.

- *collision resolution* is mapped to the *rebase* operation in Git.

The reader is supposed to be familiar with git commands and paradigms. We take as reference documentation the one available at [32].

4.2 Git workflow

The workflow shown in Figure 3.5 can be mapped to a Git workflow as follows:

- The remote repository, shown in Figure 4.1, is composed only by the *master* branch and contains *n commits*. The latest *commit*, having *commit* message G1, is purely generated.

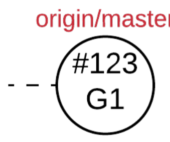


Figure 4.1: Remote repository: initial status

- A *human* developer *clones* the remote repository into a local one. As shown in Figure 4.2 there is only the *master* branch and HEAD points at the master branch.

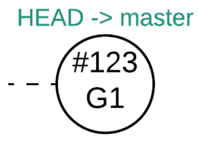


Figure 4.2: Local repository: initial status, clone of the remote repository

- The *human* developer manually introduces a new feature by applying changes to G1. She *stashes* all the changes and *commits* with message M1 (Figure 4.3).

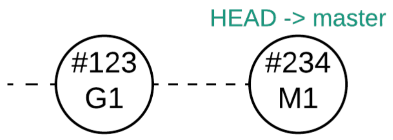


Figure 4.3: Local repository: manual changes introduced by the human developer

- The *human* developer uploads the content of the local repository to the remote repository. The remote repository (Figure 4.4) now stores the new features manually introduced by the *human* developer.

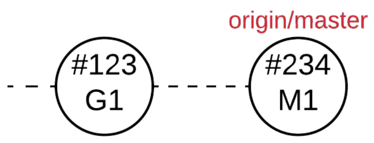


Figure 4.4: Remote repository: manual changes are safely stored

- The *human* developer deletes the local repository. Her work is safely stored in the remote repository.

- The model is updated and the transformation is ready to be executed.

- The *virtual* developer creates a local copy of the remote repository and *checkout* the *commit* having message G1. In this way the *virtual* developer aligns its local code-base to the status that reflects the current version of the model. She creates a branch named *model*, as shown in Figure 4.5.

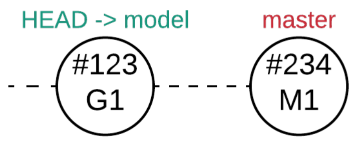


Figure 4.5: Local repository: the status reflects the current version of the model

- The *virtual* developer executes the transformation and stores the result into the repository (Figure 4.6), replacing the entire textual artifact. Then she *stashes* all changes and *commit* with message G2.

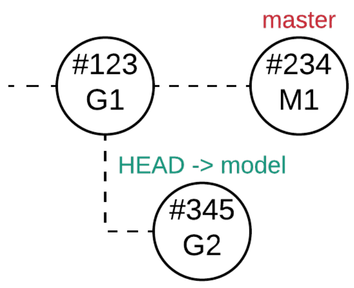


Figure 4.6: Local repository: new purely generated revision is safely stored in model branch

- In order to leave the *master* branch untouched the *virtual* developer creates a new branch *change* starting from the latest *commit* of the *master* branch.

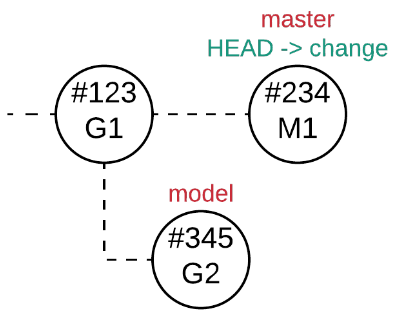


Figure 4.7: Local repository: new branch created from master branch

- A *rebase* operation is performed from the *change* branch onto the *model* branch, as shown in Figure 4.8. All the changes that the *human* developer made starting from the *commit* having message G1 are applied on top of the *model* branch, thus on top of the *commit* having message G2. During this phase conflicts may arise: some changes could not be directly applied to the *commit* having message G2 and *human* developer intervention may be needed to solve conflicts. The result is stored into the *change* branch, in a *commit* having message M1'.

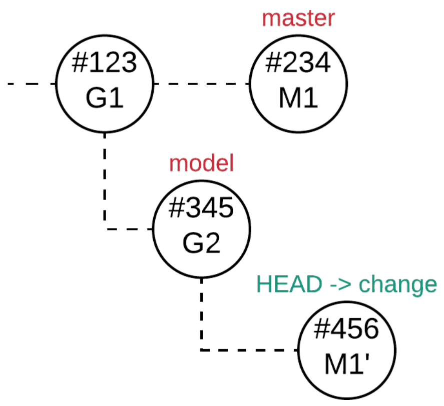


Figure 4.8: Local repository: manual changes are applied on top of model branch

- Now the *virtual* developer has to store the results onto the *master* branch in order to *push* them to the remote repository. The *virtual* developer creates a *commit* G2 on the *master* branch. The artifact represents the current status of the model. This step is necessary for subsequent alignments.

- The *virtual* developer creates a commit having message M1' on the *master* branch. In this way she stores in the *master* branch the result of the conflict resolution phase. The status of the local repository is shown in Figure 4.9.

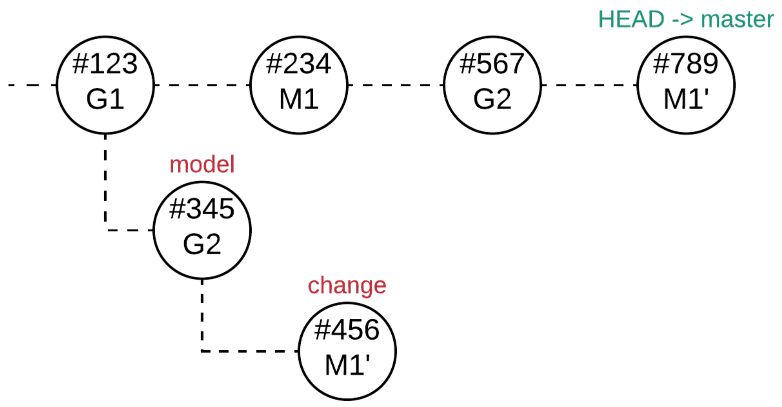


Figure 4.9: Local repository: master branch is updated with new revisions

- The *master* branch contains all the information. The *virtual* developer can now *delete* the other two branches (Figure 4.10).

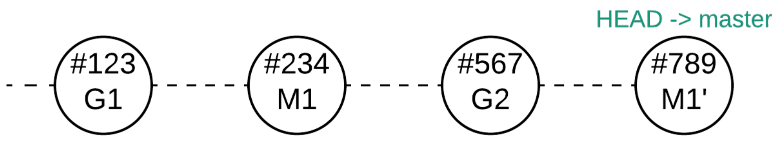


Figure 4.10: Local repository: remove the unnecessary branches

- The *virtual* developer uploads the local repository content to the remote repository. The status of the remote repository is shown in Figure 4.11.

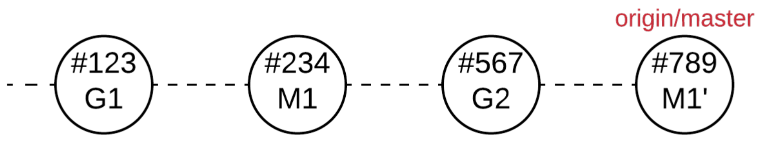


Figure 4.11: Remote repository: new purely generated revision and changes are safely stored

The *human* developer always starts working from the latest *commit*, while the *virtual* developer is always out-of-sync and starts working from the last purely generated *commit*.

4.3 Procedure automation

In this section we will present *almostjs-git* [33], a middleware system built on top of Git [27]. The proposed workflow has to be used every time a new model update is needed. The system must face two problems:

- It has to know what is the *commit* corresponding to the last generated revision.
- It has to allow the *human* developer to manually solve conflicts if needed and continue the procedure when conflicts are solved.

4.3.1 Status of the evolution process

The system must always know what is the status of the evolution process. We can identify 3 statuses of the evolution process:

1. UNDEFINED: The *commit hash* of the the *commit* corresponding to the last generated revision is not known. The evolution process cannot be executed.

2. READY: The *hash* of the the *commit* corresponding to the last generated revision is known and the evolution process has not started yet.

3. CONFLICTED: The evolution process has started but manual conflicts resolution is needed. The *human* developer can either *abort* the evolution process or resolve the conflicts and *continue* the evolution process.

A state-chart of the evolution process is shown in Figure 4.12. The status may change according to the command that the developer uses. The available commands are explained in Section 4.3.2.

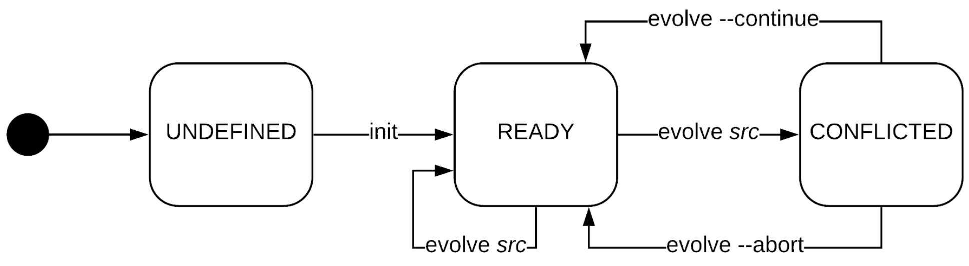


Figure 4.12: State chart of the evolution process

Status file

The *hash* of the *commit* corresponding to the last generated revision and a string representing the status of the evolution process are stored in a *json* file. The content of the file when the status is READY is shown in Listing 4.1.

```
{
    "hash": "c4a76a7",
    "status": "READY"
}
```

Listing 4.1: Status file example

4.3.2 Available commands

The *virtual* developer is provided with commands to manage the evolution process.

$ almost-git init

The *init* command is necessary to allow the *virtual* developer to start using the other commands. It is responsible of initializing the repository and the status file, setting the status to READY and saving the hash of the the *commit* corresponding to the last generated revision. A flowchart representing the operations executed by the init commands is shown in Figure 4.13.

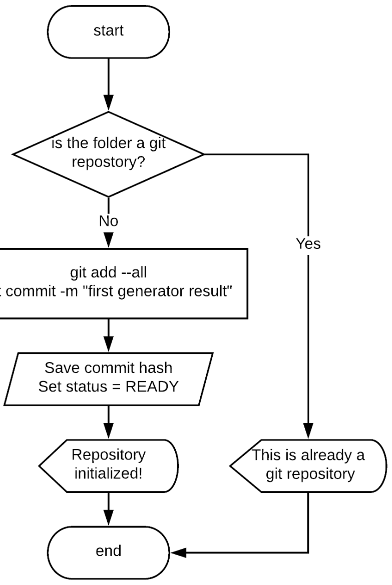

start
Is the folder a git repostory?
No
Yes
git add --all
git commit -m "first generator result"
Save commit hash
Set status = READY
Repository initialized!
This is already a git repository
end

Figure 4.13: Flowchart describing almost-git init command

$ almost-git evolve

The *evolve* command allows the *virtual* developer to start the evolution process. It reads and writes the status file in order to execute the various steps. The command accepts various options, used to drive the execution flow.

- *$ almost-git evolve [src]*

  The *virtual* developer use the parameter *src* to specify the path of the folder storing the new generated artifact. The path is needed by the system to copy the artifact into the repository when needed. This option allows the *virtual* developer to start the evolution process. Figure 4.14 shows a flowchart representing the execution flow. During the execution of this process conflicts may arise. In this case the status of the evolution process will be set to CONFLICTED and the *human* developer will be asked to manually solve the conflicts and continue the evolution process.

- *$ almost-git evolve –continue*

  Conflicts may arise during evolution process. In this case the process is interrupted and the *human* developer is asked to manually solve conflicts. Once the *human* developer solved the conflicts she uses this command to continue the evolution process. A flowchart representing the execution flow of this command is shown in Figure 4.15.

- *$ almost-git evolve –abort*

  The *human* developer may prefer to abort the evolution rather then solving conflicts. The command allows the *human* developer to abort the execution process and restore the repository status to the initial state (as if the evolution process had never started). Details of the abort execution flow are shown in Figure 4.16.

The system [33] can be easily installed via NPM [34]. It requires Git [27] and Node.JS [35] to work properly.

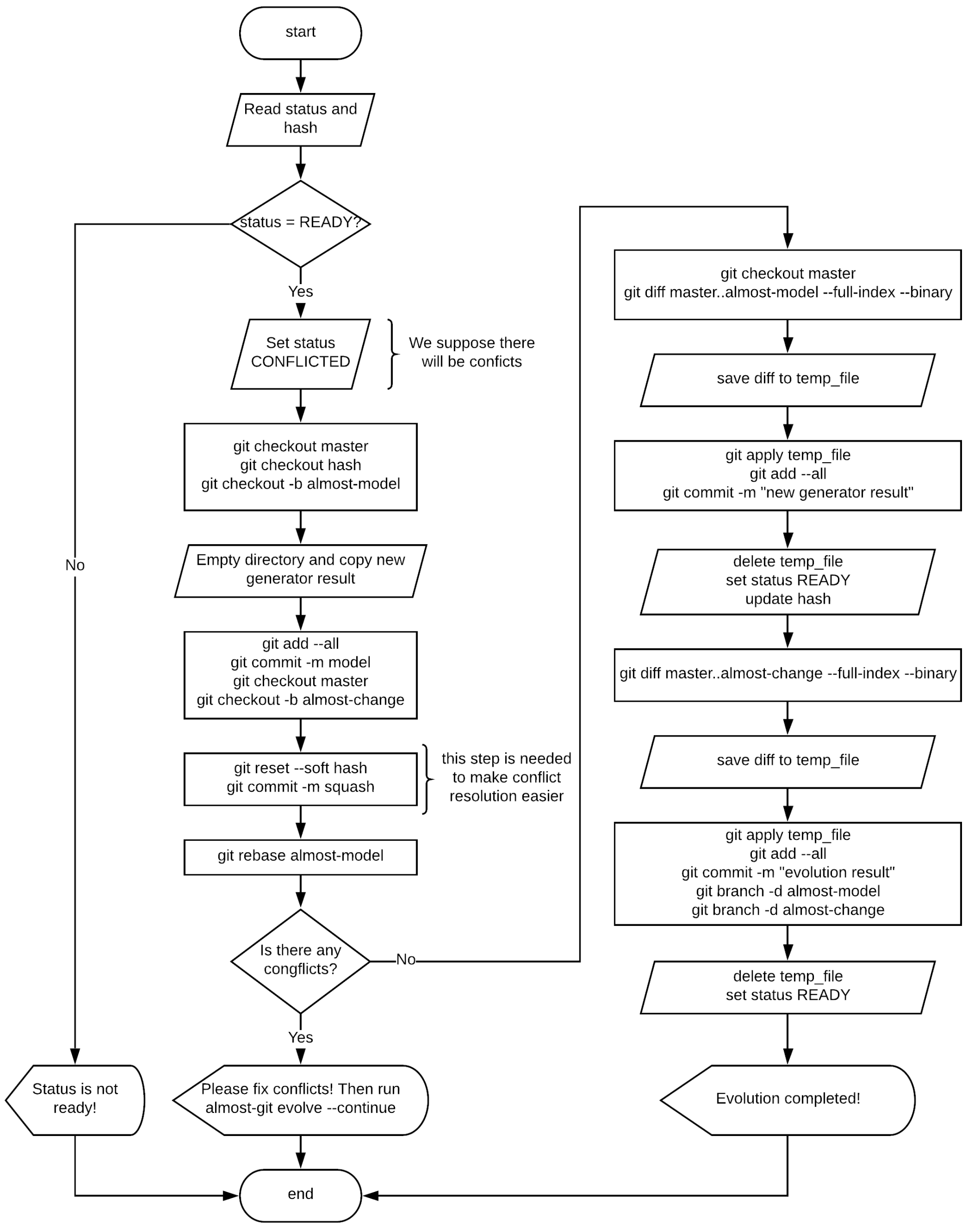


Figure 4.14: Flowchart describing almost-git evolve [src] command

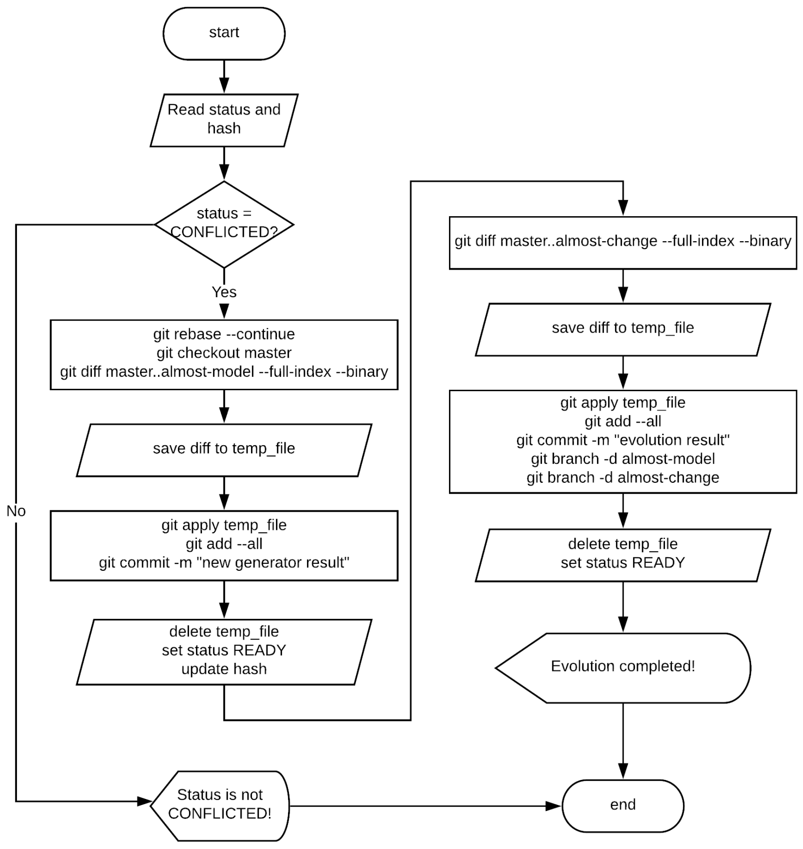


Figure 4.15: Flowchart describing almost-git evolve –continue command

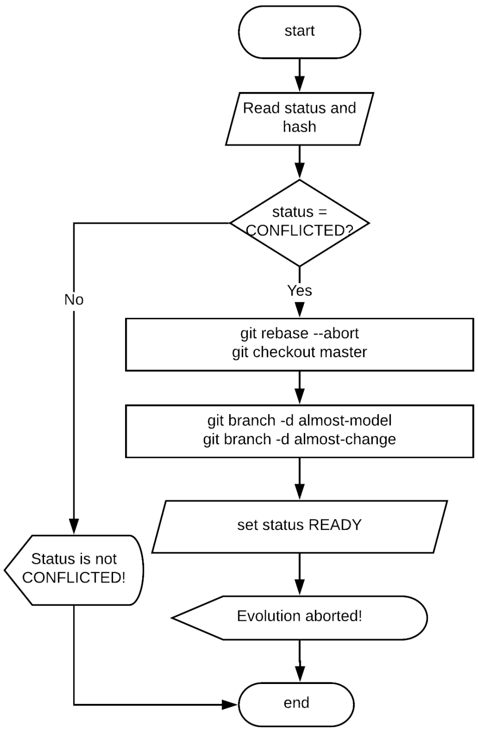


Figure 4.16: Flowchart describing almost-git evolve –abort command

Chapter 5

Experimental Study

In this Chapter we evaluate the proposed approach on the development of two applications:

- A Quiz Game, implemented as a Cordova [36] application.
- A Media Player, implemented as web application.

The evolution of the applications is described in Section 5.1. The development environments are described in Section 5.2. In Section 5.3 we compare the proposed approach with a state-of-the-art template-based forward engineering approach. In Section 5.4 we investigate on how the adoption of conflict resolution guidelines in model to text transformations may reduce the necessity of human intervention during conflict resolution. The source code of the development phases is available are available at [37].

5.1 Evolution of the applications

The applications are chosen due to their non naïve requirements: access to remote resources, multimedia and hardware capabilities of the device. The development is based on IFML (Interaction Flow Modeling Language [2]) and follows an iterative approach.

5.1.1 Quiz Game

The development follows 4 steps. At each step, new requirements are introduced, and the application model is updated. Each step also requires the enhancement of the look and feel of the application.

1. **Proof of Concept**. The application allows the user to scan a QR code (requirement coming from the game dynamics itself). Once this step is completed, the game proposes the player a multi-choice question. Once the user selects an answer, the application gives the user the correct answer and returns to the initial state.

2. **Explanation details**. The application gives the user the correct answer allows the user to investigate the explanation relative to the correct answer.

3. **Secondary Game Mechanics**. While the previous game mechanics remain available, a new one is introduced in the game. The player can choose to do not scan the QR code and receive a series of question.

4. **Internationalization**. The game mechanics are kept unchanged, but the ability to select the game language is introduced.

The IFML model is shown in Figure 5.1. Figure 5.2 shows the difference between the default UI generated by IFMLEdit.org and the final result.

5.1.2 Media Player

The second running example is a Media Player involving interaction with the media capabilities of the device and resources in the network. The development follows 4 steps:

1. **Proof of Concept**. The application loads a list of songs, allows the user to change song and pause/restart the currently playing song.

2. **Styling**. A custom style is applied to the user interface, and a system event is added to catch the end of a song.

3. **Songs Filtering**. An author filter is introduced in the application. The users can now filter the songs list in order to simplify the search.

4. **Song Cover**. The UI of the application is enhanced by showing the cover of the song currently playing.

The IFML model is shown in Figure 5.3. Figure 5.4 shows the difference between the default UI generated by IFMLEdit.org and the final result.

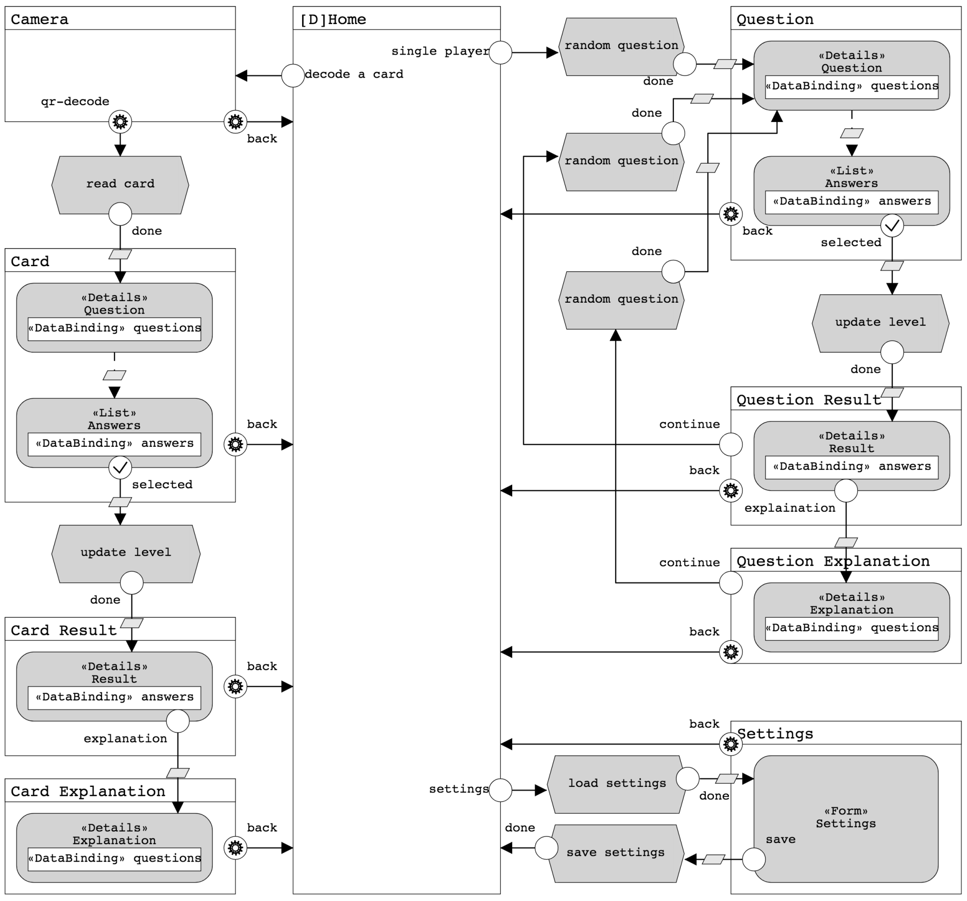
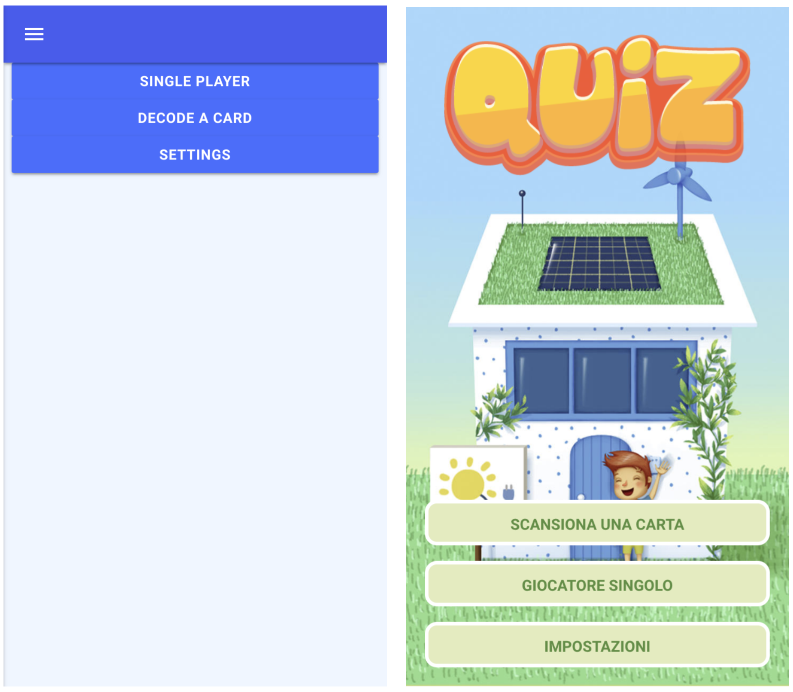


Figure 5.1: Quiz Game: IFML model



(a) Default (b) Custom

Figure 5.2: Quiz Game: from prototype to final product

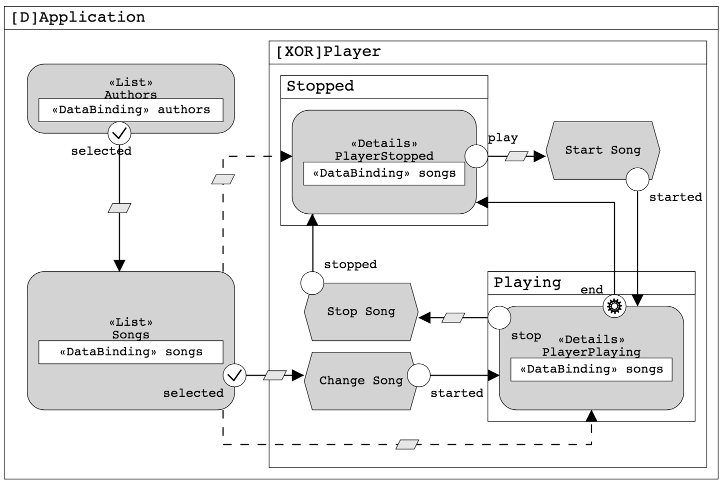
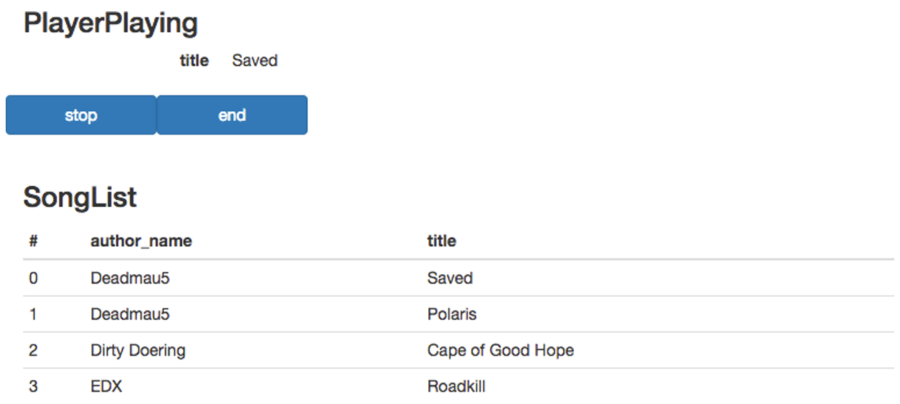


Figure 5.3: Media Player: IFML model



PlayerPlaying

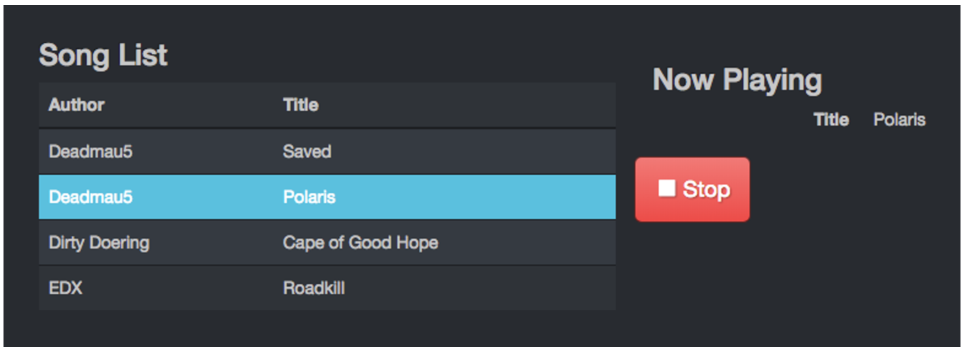
title	Saved

stop end

SongList

#	author_name	title
0	Deadmau5	Saved
1	Deadmau5	Polaris
2	Dirty Doering	Cape of Good Hope
3	EDX	Roadkill

(a) Default



Song List

Author	Title
Deadmau5	Saved
Deadmau5	Polaris
Dirty Doering	Cape of Good Hope
EDX	Roadkill

Now Playing

Title	Polaris

Stop

(b) Custom

Figure 5.4: Media Player: from prototype to final product

5.2 Development environments

The approach has been implemented and evaluated with two different development environments:

1. IFMLEdit.org [6], an open-source environment for the rapid prototyping of Web and mobile applications.

2. WebRatio, a product for the development of Web and mobile application.

In this Section we describe the two development environments.

5.2.1 IFMLEdit.org

IFMLEdit.org [6] is an online environment for the specification of IFML models and the generation of prototypes of Web and mobile applications (Figure 5.5).

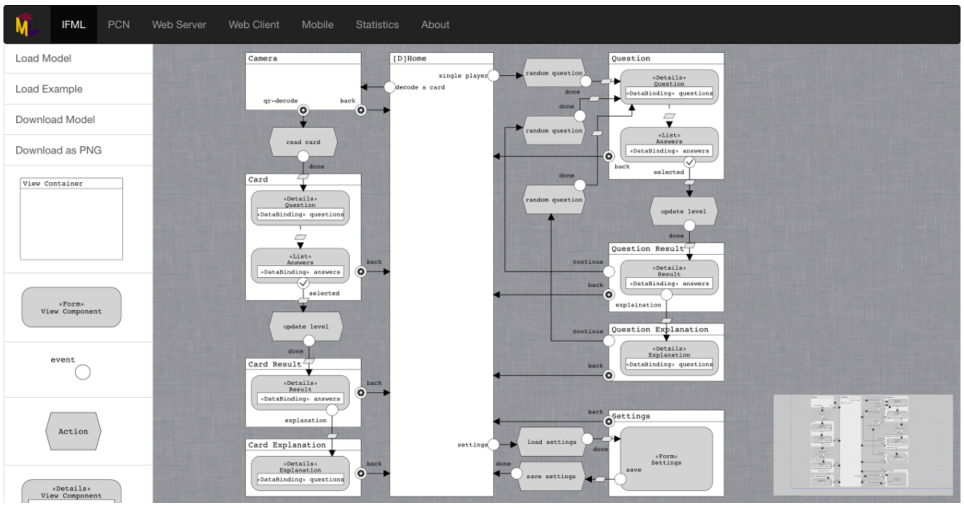


Figure 5.5: IFMLEdit.org

In IFMLEdit.org, the model of the front-end is defined with IFML, the domain model is inferred from the IFML diagram, and actions are treated as abstract black-boxes.

IFMLEdit.org is built on top of the ALMOsT.js [38] transformation framework, which allows the developer to specify model transformations with a rule-based extension of JavaScript.

Each rule is composed of two part:

1. Condition: a statement whose validity is checked for each model element.
2. Body: a function that is executed if the condition is satisfied.

```
createRule(
    function (element, model) {
        return model.isViewContainer(element);
    },
    function (container, model) {
        var children = model.getChildren(container),
            events = model.getEvents(container);
        return {
            name: 'index.html',
            content: require('./view-container-template.html.ejs')({children: children, events: events})};
        };
    )
)
```

In this example each element is checked to be of type View Container. If the condition holds the children and the events of the View Container are retrieved and passed as parameters to the template.

Each template is build with the EJS (Embedded JavaScript templates [39]) templating language. EJS requires the developers to build code skeletons to be filled with parameters.

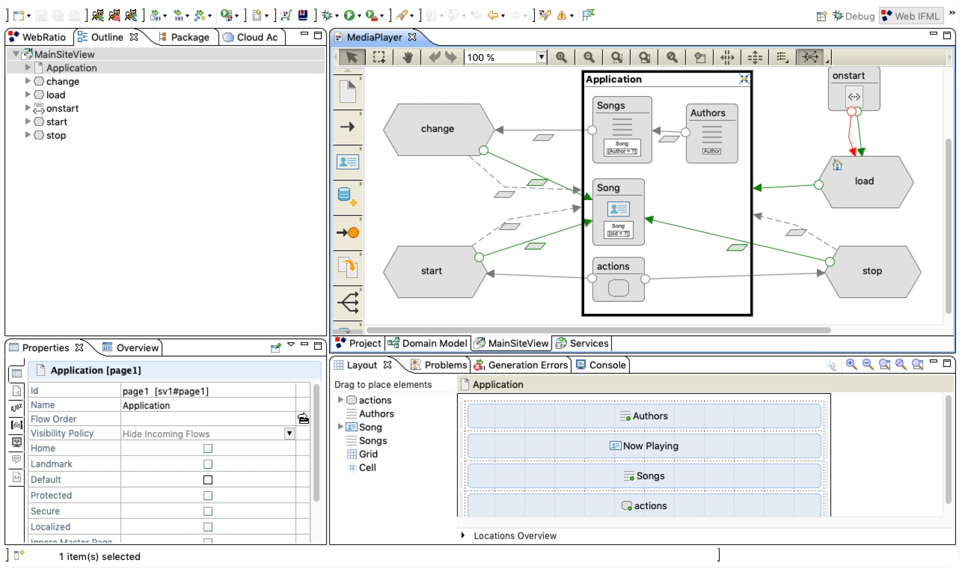
```
<span>
<% for (var i = 0; i < children.length; i += 1) { -%>
    <!-- <%= children[i].name %> -->
    <c-<%= children[i].id %> params="context: context"></c-<%= children[i].id %>>
<% }
   for (var i = 0; i < events.length; i += 1) {
       if (events[i].stereotype === 'system') { -%>
    <c-<%= events[i].id %> params="trigger: trigger.bind($data,'<%= events[i].id %>')"></c-<%= events[i].id %>>
<%     } else { -%>
    <a class="options" data-bind="click: trigger.bind($data,'<%= events[i].id %>')"><%= events[i].name %></a>
<%     }
   } -%>
</span>
```

In this example a template that generates the HTML of a View Container is shown. The children of the view container and the events related to the view container are traversed and for each of them a custom tag is generated.

IFMLEdit.org features a Web and a mobile code generator, both based on HTML, CSS and JavaScript. The former produces a client-side application, which can be connected to any preexisting REST service back-end. The latter produces a Cordova [36] application.

5.2.2 WebRatio

WebRatio is a commercial tool for the development of Web and mobile applications (Figure 5.6).



Application [page1]

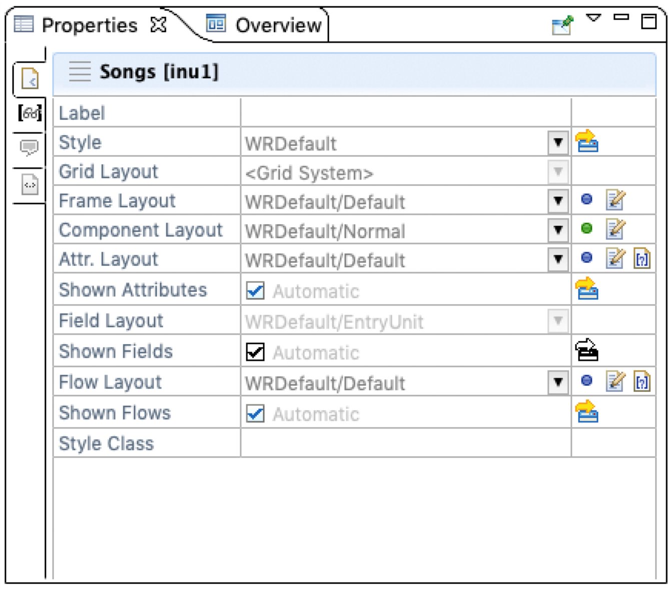
Id	page1 [sv1#page1]
Name	Application
Flow Order	
Visibility Policy	Hide Incoming Flows
Home	☐
Landmark	☐
Default	☐
Protected	☐
Secure	☐
Localized	☐

Figure 5.6: WebRatio Enterprise Platform

The tool enables the generation of the full code of the application. The user interaction is defined via IFML, the domain model via UML Class Diagrams, and the business logic via a proprietary Action Definition Language.

Details that cannot be modeled directly are incorporated using a template-based approach. An extension of IFML allows the developer to tag each IFML element with the custom template (Figure 5.7) to use for code generation.

The developer can create new custom templates or modify the existing ones. The construction of templates requires the developers to be familiar with the Groovy scripting language and WebRatio Generation Tags. WebRatio has a build-in routine that checks for inconsistencies in the Groovy code and WebRatio Generation Tags. However, it does



Songs [inu1]

Label		
Style	WRDefault	
Grid Layout	<Grid System>	
Frame Layout	WRDefault/Default	
Component Layout	WRDefault/Normal	
Attr. Layout	WRDefault/Default	
Shown Attributes	✓ Automatic	
Field Layout	WRDefault/EntryUnit	
Shown Fields	✓ Automatic	
Flow Layout	WRDefault/Default	
Shown Flows	✓ Automatic	
Style Class		

Figure 5.7: WebRatio Enterprise Platform: template selection

not imply that the generated code is valid thus the developer has to validate it manually.

5.3 Model and code co-evolution vs Template-based forward engineering

The use-case applications have been developed with both model and code co-evolution and the template-based forward engineering processes. Each sprint of the two processes shares the same phase of model editing, in which the model is updated and the code is generated. However, the two processes differ in the way in which the manually developed features are incorporated in the MDD loop. In Section 5.3.1 and Section 5.3.2 we describe the phases of the two processes focusing on the metrics used to compare the efforts. In Section 5.3.4 we provide examples to better highlight the differences in the two approaches.

5.3.1 Model and code co-evolution

Each sprint can be split in two different phases:

1. **Conflict resolution**. The previous manual changes are added on top of the generated code. The developer has to work on the parts

of the code that produce collisions. The work required to solve collision is proportional to the size of the part of the code where the automatically generated and the manually programmed code interfere. The first sprint never generates collision since there are no previous manual changes. We evaluate the effort required for collision resolution quantifying the number of collision groups (updates) and the number of lines requiring manual revision.

2. **Feature development**. The developer is free to manually change the resulting code in order to implement the desired features. The work required to manually introduce features is quantified in terms of the updates needed to build them and of the lines of code affected.

The total application development effort can be obtained by adding up the values related to conflict resolution phase and those related to feature development phase.

5.3.2 Template-based forward engineering

Also the template-based forward engineering process can be split in two different phases:

1. **Feature experimentation**. The developer is free to manually experiment on the generated code in order to implement the desired changes. The work required to manually experiment features is quantified in terms of the updates needed to build them and of the lines of code affected.

2. **Back-porting**. The developer updates the templates of the code generator, so that manually programmed features are automatically reproduced in the subsequent code generation steps. The work required by the back-porting process is proportional to the code-level size of manual changes made by developers. The effort of this phase is represented by the number of updates required to obtain the templates and the number of lines involved.

The total application development effort can be obtained by adding up the values related to feature experimentation phase and those related to feature experimentation phase.

5.3.3 Experimental results

The use-case applications have been developed with both IFMLEdit.org and Webratio.

IFMLEdit.org

Table 5.1 and 5.2 show respectively the result of template-based development process and model and code co-evolution development process of Quiz Game application. Table 5.3 and 5.4 show respectively the result of template-based development process and model and code co-evolution development process of Media Player application.

Table 5.1, 5.2, 5.3 and 5.4 show also the total application **development** work. The model and code co-evolution process required less than 75% of the effort required by the template-based forward engineering process. In particular:

- The number of updates required was reduced by 27% in the Quiz Game and by 28% in the Media Player.
- The number of lines of code was reduced by 27% both in the Quiz Game and in the Media Player

The efforts required by feature development and feature experimentation are comparable, but the work required by the conflict resolution phase is less than 20% of that required by back-porting of features into templates.

The difference in the total development effort is due to the overhead of template programming, which requires extra code for integrating the feature into the code generator. This is especially true in the development of GUI templates, where achieving the desired visual effect is easier in the actual implementation code than in the code of the template.

WebRatio

Table 5.5 and 5.6 report the amount of work respectively required by collision resolution and back-porting during the development of the Quiz Game application. Table 5.7 and 5.8 report the amount of work respectively required by collision resolution and back-porting during the development of the Media Player application.

Sprint	Model & Code Co-evolution			
	Feature Development		Conflict resolution	
	Updates	Atomic Updates	Updates	Atomic Updates
Proof of Concept	27	1154	0	0
Styling and Expl.	9	111	4	77
2^{nd} Mechanics	29	492	3	20
I18n	33	237	5	139
Total	98	1994	12	236

	Total effort (development + integration)	
	Updates	Atomic updates
	110	2230

Table 5.1: Quiz Game: Model & Code Co-evolution development statistics with IFMLEdit.org

Sprint	Template-based forward engineering			
	Feature Experimentation		Back-porting	
	Updates	Atomic Updates	Updates	Atomic Updates
Proof of Concept	28	1110	27	1102
Styling and Expl.	6	50	6	66
2^{nd} Mechanics	18	255	15	127
I18n	27	181	24	179
Total	79	1596	72	1474

	Total effort (experimentation + back-porting)	
	Updates	Atomic updates
	151	3070

Table 5.2: Quiz Game: Template-based forward engineering development statistics with IFMLEdit.org

Sprint	Model & Code Co-Evolution			
	Feature Development		Conflict resolution	
	Updates	Atomic Updates	Updates	Atomic Updates
Proof of Concept	16	144	0	0
Styling	5	32	1	3
Songs Filtering	4	47	1	4
Songs Cover	6	34	2	10
Total	31	257	4	17
	Total effort (development + integration)			
	Updates		Atomic updates	
	35		274	

Table 5.3: Media Player: Model & Code Co-Evolution development statistics with IFMLEdit.org

Sprint	Template-based forward engineering			
	Feature Experimentation		Back-porting	
	Updates	Atomic Updates	Updates	Atomic Updates
Proof of Concept	12	100	15	148
Styling	5	42	5	32
Songs Filtering	3	17	4	16
Songs Cover	3	14	2	9
Total	23	173	26	205
	Total effort (experimentation + back-porting)			
	Updates		Atomic updates	
	49		378	

Table 5.4: Media Player: Template-based forward engineering development statistics with IFMLEdit.org

Table 5.5, 5.6, 5.7 and 5.8 show the total application **development** work. The model and code co-evolution process required less than 65% of the effort required by the template-based forward engineering process. In particular:

- The number of updates required was reduced by 45% in the Quiz Game and by 35% in the Media Player.
- The number of lines of code was reduced by 38% in the Quiz Game and by 67% in the Media Player.

Given the complexity and expressive power of the WebRatio template language, the same feature required a considerably higher work for back-porting than for conflict resolution. The efforts required by feature development and feature experimentation are comparable, but the work required by conflict resolution is less than 20% of that required by back-porting of features into templates. The effort for back-porting is comparable to the whole effort required by model and code co-evolution.

5.3.4 Example

In this Section, we show an example of how the collision between the Virtual Developer and the Human Developer is detected and solved and how the same customization is achieved by templating. The example is taken from the Quiz Game application development with IFMLEdit.org, from the second to the third version, and limited to the HTML code representing the Home View Container.

Model and code co-evolution

The second sprint starts with the automatic generation of code from the model. The only events related to the Home View Container is *decode a card*. The code generated by the Virtual Developer contains the following fragment, which produces the default rendition of a menu element, shown in figure 5.2.

```
<span>
    <a class="btn cyan col s12" data-bind="click: trigger.bind(
        $data,'event-home-decode-card')"> decode a card</a>
</span>
```

Sprint	Model & Code Co-evolution			
	Feature Development		Conflict resolution	
	Updates	Atomic Updates	Updates	Atomic Updates
Proof of Concept	5	142	0	0
Styling and Expl.	4	29	0	0
2^{nd} Mechanics	10	42	0	0
I18n	5	15	0	0
Total	24	228	0	0

	Total Effort (development + integration)	
	Updates	Atomic Updates
	24	228

Table 5.5: Quiz Game: Model & Code Co-evolution development statistics with WebRatio

Sprint	Template-based forward engineering			
	Feature Experimentation		Back-porting	
	Updates	Atomic Updates	Updates	Atomic Updates
Proof of Concept	5	136	13	137
Styling and Expl.	4	24	4	22
2^{nd} Mechanics	4	11	4	8
I18n	5	16	5	16
Total	18	187	26	183

	Total Effort (experimentation + back-porting)	
	Updates	Atomic Updates
	44	370

Table 5.6: Quiz Game: Template-based forward engineering development statistics with WebRatio

Sprint	Model & Code Co-Evolution			
	Feature Development		Conflict resolution	
	Updates	Atomic Updates	Updates	Atomic Updates
Proof of Concept	2	21	0	0
Styling	6	47	0	0
Songs Filtering	2	4	2	7
Songs cover	1	2	0	0
Total	11	74	2	7

	Total Effort (development + integration)	
	Updates	Atomic Updates
	13	81

Table 5.7: Media Player: Model & Code Co-Evolution development statistics with WebRatio

Sprint	Template-based forward engineering			
	Feature Experimentation		Back-porting	
	Updates	Atomic Updates	Updates	Atomic Updates
Proof of Concept	2	21	4	25
Styling	5	48	6	153
Songs Filtering	0	0	0	0
Songs cover	1	2	2	4
Total	8	71	12	182

	Total Effort (experimentation + back-porting)	
	Updates	Atomic Updates
	20	253

Table 5.8: Media Player: Template-based forward engineering development statistics with WebRatio

The human developer modifies the code to apply a custom style to the menu item. Her submission is accepted and the following delta is computed (- denotes line deletion, + line addition).

```
<span>
-    <a class="btn cyan col s12" data-bind="click: trigger.bind(
     $data,'event-home-decode-card')">decode a card</a>
+    <a class="options" data-bind="click: trigger.bind($data,'
     event-home-decode-card')">decode a card</a>
</span>
```

The third sprint starts and the model is updated. The secondary game mechanic is introduced and the event *single player* is added to the Home View Container. The Virtual Developer generates the code again, producing the following delta:

```
<span>
+    <a class="btn cyan col s12" data-bind="click: trigger.bind(
     $data,'event-home-single-player')"> single player</a>
     <a class="btn cyan col s12" data-bind="click: trigger.bind(
         $data,'event-home-decode-card')"> decode a card</a>
</span>
```

The code generator reproduces menu elements with the default presentation, overwriting the manual changes. Submitting the generated code to the central repository a collision is signaled by the Git system.

```
<span>
<<<<<<< HEAD
     <a class="btn cyan col s12" data-bind="click: trigger.bind(\
         $data,'event-home-single-player')"> single player</a>
     <a class="btn cyan col s12" data-bind="click: trigger.bind(\
         $data,'event-home-decode-card')"> decode a card</a>
=======
     <a class="options" data-bind="click: trigger.bind(\$data,'
         event-home-decode-card')"> decode a card</a>
>>>>>>> squash commits
</span>
```

The human developer resolves the collision producing the following code.

```
<span>
     <a class="options" data-bind="click: trigger.bind(\$data,'
         event-home-decode-card')"> decode a card</a>
     <a class="btn cyan col s12" data-bind="click: trigger.bind(\
         $data,'event-home-single-player')"> single player</a>
</span>
```

The human developer can proceed customizing the presentation of the *single player* event too.

Template-based forward engineering

Customizing the presentation of the interface can also be achieved with a template-based approach. In order to apply the custom presentation to all the events associated with an IFML ViewContainer, the code generator of IFMLEdit.org has been modified, obtaining the following template:

```
<span>
<% for (var i = 0; i < children.length; i += 1) { -%>
    <!-- <%= children[i].name %> -->
    <c-<%= children[i].id %> params="context: context"></c-<%= children[i].id %>>
<% }
   for (var i = 0; i < events.length; i += 1) {
       if (events[i].stereotype === 'system') { -%>
    <c-<%= events[i].id %> params="trigger: trigger.bind($data,'<%= events[i].id %>')"></c-<%= events[i].id %>>
<%     } else { -%>
    <a class="options" data-bind="click: trigger.bind($data,'<%= events[i].id %>')"><%= events[i].name %></a>
<%     }
   } -%>
</span>
```

5.4 Evaluation of Collision Prevention Guidelines

Table 5.1 and 5.3 show that conflict resolution is accountable for 11% of the total application development effort (10.9% for the Quiz Game and 11.4% for the Media Player). We were able to identify various collisions that could be avoided; to do so, the original code generator of IFMLEdit.org has been compared to a new version implementing the guidelines discussed in Section 3.4.

```
<span>
<% for (var i = 0; i < children.length; i += 1) { -%>
    <!-- <%= children[i].name %> -->
    <c-<%= children[i].id %> params="context: context"></c-<%= children[i].id %>>
<% }
   for (var i = 0; i < events.length; i += 1) {
       if (events[i].stereotype === 'system') { -%>
    <c-<%= events[i].id %> params="trigger: trigger.bind($data
        ,'<%= events[i].id %>')"></c-<%= events[i].id %>>
<%      } else { -%>
    <a class="options" data-bind="click: trigger.bind($data,'<%=
        events[i].id %>')"><%= events[i].name %></a>
<%      }
   } -%>
</span>
```

The template above, included in the code generator of IFMLEdit.org, generates HTML file from View Container elements. The template has been modified in order to ensure deterministic iteration over the children and events of the view Container. Moreover, line level separation of concerns is introduced.

```
<span>
<%  var _children = children.sort
    for (var i = 0; i < children.length; i+=1) { -%>
    <!-- <%= children[i].name %> -->
    <c-<%= children[i].id %> params="context: context">
    </c-<%= children[i].id %>>
<%  } -%>
<% for (var i = 0; i < events.length; i += 1) {
        if (events[i].stereotype == 'system') { -%>
  <c-<%= events[i].id %> params="trigger: trigger.bind($data
      ,'<%= events[i].id %>')">
  </c-<%= events[i].id %>>
<%      } else { -%>
  <a class="col-xs-2 btn btn-primary"
    data-bind="click: trigger.bind($data,'<%= events[i].id %>')
        ">
    <%= events[i].name %>
  </a>
<%      }
   } -%>
</span>
```

Such avoidable collisions are responsible for the majority of the updates and affected lines involved in collision resolution.

We developed the two use case applications with both the original and for the enhanced version of the IFMLEdit.org code generator. For each sprint we collected the following data:

1. The work required by the Human Developer to manually introduce features, quantified in terms of the updates needed to build them and of the lines of code affected.

2. The changes introduced by the Virtual Developer w.r.t. the last purely generated revision, quantified in terms of the updates needed to build them and of the lines of code affected.

3. The effort required for collision resolution quantifying the number of collision groups(updates) and the number of lines requiring manual revision.

Table 5.9 and 5.10 show the data collected during the development of the Quiz Game application.

Table 5.12 and Table 5.11 show the data collected for the Media Player application.

The number of updates of the virtual developer depends on the magnitude of the changes on the model at each sprint and is independent from previous sprints. The number of updates of the human developer grows because at each sprint the new manual changes add up to the ones of previous sprints.

It can be noted that the implementation of the collision prevention guidelines in the code generator reduces the number of collisions. In particular:

- The number of collision groups is reduced by 75% in the Quiz Game and by 60% in the Media Player.

- The number of lines affected is reduced by 79% in the Quiz Game and by 27% in the Media Player.

For evaluating the impact of the improved code generator on the total development effort, only the number of updates can be used, because the number of lines increases when the line separation rules are added to the model transformation.

The collision prevention guidelines reduce the total development effort by 8% in the Quiz Game and 7% in the Media Player.

These results show that even very simple design rules for the code transformation can increase automatic resolution and thus reduce the collision management effort.

Sprint	Original Implementation					
	Human Dev.		Virtual Dev.		Collisions	
	Updates	Atomic Up.	Updates	Atomic Up.	Updates	Atomic Up.
Proof of Concept	-	-	-	1442	-	-
Styling and Expl.	27	1124	21	287	4	77
2^{nd} Mechanics	30	1196	45	1168	3	20
I18n	47	1612	27	457	5	139
				Total	12	236

Table 5.9: Quiz Game: collisions with collision prevention rules

Sprint	Collision Prevention					
	Human Dev.		Virtual Dev.		Collisions	
	Updates	Atomic Up.	Updates	Atomic Up.	Updates	Atomic Up.
Proof of Concept	-	-	-	1510	-	-
Styling and Expl.	27	1054	16	274	1	37
2^{nd} Mechanics	30	1235	44	1209	1	6
I18n	47	1709	25	404	1	6
				Total	3	49

Table 5.10: Quiz Game: collisions without collision prevention rules

Sprint	Original Implementation					
	Human Dev.		Virtual Dev.		Collisions	
	Updates	Atomic Up.	Updates	Atomic Up.	Updates	Atomic Up.
Proof of Concept	-	-	-	1202	-	-
Styling	14	141	9	93	1	3
Songs Filtering	18	178	14	258	1	4
Songs Cover	22	217	12	303	2	10
				Total	5	17

Table 5.11: Media Player: collisions without collision prevention rules

Sprint	Collision Prevention					
	Human Dev.		Virtual Dev.		Collisions	
	Updates	Atomic Up.	Updates	Atomic Up.	Updates	Atomic Up.
Proof of Concept	-	-	-	1249	-	-
Styling	16	144	6	71	0	0
Songs Filtering	20	176	11	238	0	0
Songs Cover	24	223	5	268	2	14
				Total	2	14

Table 5.12: Media Player: collisions with collision prevention rules

Chapter 6

Conclusions and Future Work

We presented an approach for model and text co-evolution which considers the code generator as a Virtual Developer and apply the same version control techniques normally used in distributed development.

The approach was evaluated on the development of two use-cases applications. The assessment shows that, in IFMLEdit.org, model and code co-evolution process requires, in the worst case scenario, **27%** less code updates and impacts **27%** less lines than the template based approach; in WebRatio, model and code co-evolution requires, in the worst case, **35%** less code updates and impacts **38%** less lines than the template based approach.

The approach was used in the MDD development of energy awareness digital game [40, 41] using IFMLEdit.org as code generator. The presentation code and the code for connecting the game to a back-end cloud platform are added manually. While the described approach 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 research of language dependent collision resolution strategies to understand if their adoption can lower the impact of conflict resolution phase on the total development effort. Moreover, a large scale experimental study is required in order to further investigate the comparison of the proposed approach with the template based one. Experimentation and further assessment of the proposed approach in the industry are also required, 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.

Bibliography

[1] UML unified modeling language. `www.uml.org/`. Accessed: 2018-02-20.

[2] OMG. Interaction flow modeling language (IFML), version 1.0. `http://www.omg.org/spec/IFML/1.0/`, 2015.

[3] Carlo Bernaschina, Sara Comai, and Piero Fraternali. Formal semantics of omg's interaction flow modeling language (IFML) for mobile and rich-client application model driven development. *Journal of Systems and Software*, 137:239–260, 2018.

[4] Sara Comai and Piero Fraternali. A semantic model for specifying data-intensive web applications using webml. In Isabel F. Cruz, Stefan Decker, Jérôme Euzenat, and Deborah L. McGuinness, editors, *Proceedings of SWWS'01, The first Semantic Web Working Symposium, Stanford University, California, USA, July 30 - August 1, 2001*, pages 566–585, 2001.

[5] Roberto Rodríguez-Echeverria, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Towards a UML and IFML mapping to graphql. In Irene Garrigós and Manuel Wimmer, editors, *Current Trends in Web Engineering - ICWE 2017 International Workshops, Liquid Multi-Device Software and EnWoT, practi-O-web, NLPIT, SoWeMine, Rome, Italy, June 5-8, 2017, Revised Selected Papers*, volume 10544 of *Lecture Notes in Computer Science*, pages 149–155. Springer, 2017.

[6] Carlo Bernaschina, Sara Comai, and Piero Fraternali. IFMLEdit.org: Model driven rapid prototyping of mobile apps. In *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 207–208. IEEE, 2017.

[7] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, Abel Gómez, Massimo Tisi, and Jordi Cabot. EMF-REST: generation of restful apis from models. In Sascha Ossowski, editor, *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, pages 1446–1453. ACM, 2016.

[8] Roberto Acerbis, Aldo Bongio, Stefano Butti, Stefano Ceri, Fulvio Ciapessoni, Carlo Conserva, Piero Fraternali, and Giovanni Toffetti. Webratio, an innovative technology for web application development. In Nora Koch, Piero Fraternali, and Martin Wirsing, editors, *Web Engineering - 4th International Conference, ICWE 2004, Munich, Germany, July 26-30, 2004, Proceedings*, volume 3140 of *Lecture Notes in Computer Science*, pages 613–614. Springer, 2004.

[9] Martin Henkel and Janis Stirna. Pondering on the key functionality of model driven development tools: The case of mendix. In Peter Forbrig and Horst Günther, editors, *Perspectives in Business Informatics Research - 9th International Conference, BIR 2010, Rostock Germany, September 29-October 1, 2010. Proceedings*, volume 64 of *Lecture Notes in Business Information Processing*, pages 146–160. Springer, 2010.

[10] Outsystems development environment. www.outsystems.com/. Accessed: 2018-02-20.

[11] Zoho creator. www.zoho.com/creator/. Accessed: 2018-02-20.

[12] Santiago P. Jácome-Guerrero and Juan de Lara. Controlling metamodel extensibility in model-driven engineering. *IEEE Access*, 6:19923–19939, 2018.

[13] Hugo Brunelière, Jokin García, Philippe Desfray, Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, and Jordi Cabot. On lightweight metamodel extension to support modeling tools agility. In Gabriele Taentzer and Francis Bordeleau, editors, *Modelling Foundations and Applications - 11th European Conference, ECMFA 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-24, 2015. Proceedings*, volume 9153 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 2015.

[14] Zhiyi Ma, Huihong He, Jinyang Liu, and Xiao He. Analysis of the evolution of the UML metamodel. In Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic, editors, *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018.*, pages 356–363. SciTePress, 2018.

[15] Andreas Demuth, Markus Riedl-Ehrenleitner, Roberto E. Lopez-Herrejon, and Alexander Egyed. Co-evolution of metamodels and models through consistent change propagation. *Journal of Systems and Software*, 111:281–297, 2016.

[16] Andreas Demuth, Roberto E. Lopez-Herrejon, and Alexander Egyed. Co-evolution of metamodels and models through consistent change propagation. In Alfonso Pierantonio and Bernhard Schätz, editors, *Proceedings of the Workshop on Models and Evolution co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, Florida, USA, October 1, 2013.*, volume 1090 of *CEUR Workshop Proceedings*, pages 14–21. CEUR-WS.org, 2013.

[17] Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. Approaches to co-evolution of metamodels and models: A survey. *IEEE Trans. Software Eng.*, 43(5):396–414, 2017.

[18] Jokin García, Oscar Díaz, and Jordi Cabot. An adapter-based approach to co-evolve generated SQL in model-to-text transformations. In Matthias Jarke, John Mylopoulos, Christoph Quix, Colette Rolland, Yannis Manolopoulos, Haralambos Mouratidis, and Jennifer Horkoff, editors, *Advanced Information Systems Engineering - 26th International Conference, CAiSE 2014, Thessaloniki, Greece, June 16-20, 2014. Proceedings*, volume 8484 of *Lecture Notes in Computer Science*, pages 518–532. Springer, 2014.

[19] Klaus Birken. Building code generators for dsls using a partial evaluator for the xtend language. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, volume 8802 of *Lecture Notes in Computer Science*, pages 407–424. Springer, 2014.

[20] Gábor Kövesdán, Márk Asztalos, and László Lengyel. Polymorphic templates: A design pattern for implementing agile model-to-text transformations. In Davide Di Ruscio, Juan de Lara, and Alfonso Pierantonio, editors, *Proceedings of the 3rd Workshop on Extreme Modeling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems, XM@MoDELS 2014, Valencia, Spain, September 29, 2014.*, volume 1239 of *CEUR Workshop Proceedings*, pages 32–41. CEUR-WS.org, 2014.

[21] Bernhard Hoisl and Stefan Sobernig. Towards benchmarking evolution support in model-to-text transformation systems. In Jürgen Dingel, Sahar Kokaly, Levi Lucio, Rick Salay, and Hans Vangheluwe, editors, *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), Ottawa, Canada, September 28, 2015.*, volume 1500 of *CEUR Workshop Proceedings*, pages 16–25. CEUR-WS.org, 2015.

[22] Anthony Anjorin, Marius Paul Lauder, Michael Schlereth, and Andy Schürr. Support for bidirectional model-to-text transformations. *ECEASST*, 36, 2010.

[23] Thomas Goldschmidt and Axel Uhl. Retainment policies - A formal framework for change retainment for trace-based model transformations. *Information & Software Technology*, 55(6):1064–1084, 2013.

[24] Acceleo. `https://www.eclipse.org/acceleo/`. Accessed: 2018-02-20.

[25] Framework 7 full featured html framework for building ios and android apps. `framework7.io/`. Accessed: 2018-02-20.

[26] Flutter build beautiful native apps in record time. `flutter.io/`. Accessed: 2018-02-20.

[27] Git. `https://git-scm.com/`. Accessed: 2018-02-20.

[28] Mercurial. `https://www.mercurial-scm.org/`. Accessed: 2018-02-20.

[29] Apache Subversion. `https://subversion.apache.org/`. Accessed: 2018-02-20.

[30] Susan Horwitz, Jan Prins, and Thomas W. Reps. Integrating non-interfering versions of programs. In Jeanne Ferrante and P. Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 133–145. ACM Press, 1988.

[31] Yuichi Nishimura and Katsuhisa Maruyama. Supporting merge conflict resolution by using fine-grained code change history. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 661–664. IEEE Computer Society, 2016.

[32] Git documentation. `https://git-scm.com/doc/`. Accessed: 2018-02-20.

[33] almost-git a tool for model and text co-evolution. `npmjs.com/package/almost-git`. Accessed: 2018-07-20.

[34] NPM the package manager for node.js. `npmjs.com/`. Accessed: 2018-07-20.

[35] Node.JS a javascript runtime built on chrome's v8 javascript engine. `nodejs.org/`. Accessed: 2018-07-20.

[36] Cordova. `cordova.apache.org/`. Accessed: 2018-07-20.

[37] Emanuele Falzone Master Thesis Repository. `github.com/emanuele-falzone-master-thesis/`. Accessed: 2018-11-09.

[38] Carlo Bernaschina. ALMOsT.js: An agile model to model and model to text transformation framework. In Jordi Cabot, Roberto De Virgilio, and Riccardo Torlone, editors, *Web Engineering - 17th International Conference, ICWE 2017, Rome, Italy, June 5-8, 2017, Proceedings*, volume 10360 of *Lecture Notes in Computer Science*, pages 79–97. Springer, 2017.

[39] Embedded javascript templates. `ejs.co/`. Accessed: 2018-02-20.

[40] Funergy the encompass game for energy education (android). play.google.com/store/apps/details?id=com.eu.myfunergy. Accessed: 2018-07-20.

[41] Funergy the encompass game for energy education (ios). itunes.apple.com/it/app/funergy/id1423062333?l=en&mt=8. Accessed: 2018-07-20.