The Virtual Developer: Integrating Code Generation and Manual Development with Conflict Resolution

CARLO BERNASCHINA, EMANUELE FALZONE, PIERO FRATERNALI, and SERGIO LUIS HERRERA GONZALEZ, Dipartimento di Elettronica, Informazione e Bioingegneria - Politecnico di Milano, Italy

Model Driven Development (MDD) requires proper tools to derive the implementation code from the application models. However, the integration of handwritten and generated code is a long-standing issue, which affects the adoption of MDD in the industry. This paper presents a model and code co-evolution approach that addresses such a problem a posteriori, using the standard collision detection capabilities of Version Control Systems to support the semi-automatic merge of the two types of code. We assess the proposed approach by contrasting it with the more traditional template-based, forward engineering process, adopted by most MDD tools.

CCS Concepts: • **Software and its engineering → Software development techniques**; **Software evolution**; **Software version control**; *Collaboration in software development*; Agile software development;

Additional Key Words and Phrases: Model Driven Development, Code Generation, Agile Development

ACM Reference Format:
Carlo Bernaschina, Emanuele Falzone, Piero Fraternali, and Sergio Luis Herrera Gonzalez. 2010. The Virtual Developer: Integrating Code Generation and Manual Development with Conflict Resolution. *ACM Trans. Softw. Eng. Methodol.* 9, 4, Article 39 (March 2010), 37 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Model Driven Development (MDD) advocates the use of *models* and of *model transformations* as key ingredients of software development [Stahl and Völter 2006]. Abstraction is the most important aspect of MDD. It enables developers to validate high level concepts and introduce low-level details at later stages in the process. In the *forward engineering* approach [Brambilla et al. 2017; Pleuss et al. 2013; Uhl 2008], details are introduced via model transformations, which iteratively refine the model, eventually getting to the final product. After a model change, the process is reiterated to produce a new version. Refinement, i.e., the progressive incorporation of detail, is obtained by transformations having a high-level meta-model as the source and a lower-level meta-model as the target. The source meta-model may not contain enough information to drive the production of an immediately usable output, due to its abstraction that omits details of secondary importance [Kelly and Tolvanen 2008]. A pure forward engineering approach requires enhancing the source meta-model and/or the transformation, to remove uncertainties and derive the final code from the model. However, such an approach can lead to loss of abstraction in the source model and diminishes the benefits of MDD.

Authors' address: Carlo Bernaschina; Emanuele Falzone; Piero Fraternali; Sergio Luis Herrera Gonzalez, Dipartimento di Elettronica, Informazione e Bioingegneria - Politecnico di Milano, Piazza L. Da Vinci, 21. Milano, I-20133, Italy, carlo.bernaschina@polimi.it, emanuele.falzone@mail.polimi.it, piero.fraternali@polimi.it, sergioluis.herrera@polimi.it.

For transformations to generate specific features of the output not inferable from the input model, **template-based** approaches can be used [Pleuss et al. 2013; Syriani et al. 2018]; commercial tools (e.g. WebRatio[1], Mendix[2], Outsystems[3] and Zoho Creator[4]) require developers to annotate models with tool-specific attributes, so that the transformation can select the proper templates and create the desired output. This approach has been successfully adopted in the industry, but the construction of templates could demand more work than the manual coding of the final artifact.

The enrichment of the input meta-model and the use of templates in transformations aim at generating the full implementation of the application. An alternative approach is the integration of handwritten and generated code, pursued by means of patterns that enforce the *separation* between the two types of code. Such patterns can be language-dependent (e.g., proper use of inheritance and interfaces in object-oriented languages), or general (e.g., the use of mechanisms such as protected areas and included files [Greifenberg et al. 2015; Völter and Bettin 2004]). These patterns divide the application code into automatically generated and handwritten parts, which are kept separate. Meta-model extensions, template-based transformations, and handwritten code integration patterns share the common trait that *the generated parts should not be modified by developers. The rationale of such restriction is that any manual modification would be overwritten by subsequent re-generations of the code from the model.*

In this paper, we address the question of how to manage non-modeled features in MDD and, more specifically, of how to integrate the code produced by the generator and that handwritten by the developer to implement the non-modeled features. We use as examples Web and mobile applications, where the separation of handwritten and generated code helps factoring out manually crafted presentation aspects from automatically generated interface organization and data extraction logic. We explore an approach alternative to the strict separation of handwritten and generated code: letting the developer *modify the generated code* and then *identifying and resolving collisions* between the handwritten and the generated parts. Doing so requires a proper development work-flow, which we call *model and code co-evolution work-flow*. At the heart of it, there is the idea of keeping track of the *incremental modifications* of the code-base produced by the code generator: this ensures that the analysis of the conflicts needs to compare only the *delta* produced by the last execution of the code generator and the modifications made by the programmer to the generated code; if the proposed work-flow is followed, then simple version control techniques can support the partial automation of conflict resolution. However, this task becomes the bottleneck of the MDD process and thus the question arises whether the proposed approach is practical or conflict resolution demands too much work; to understand this aspect, we compare the work required by the model and code co-evolution approach to that of a widely used industrial MDD process: template-based forward engineering. As a side observation, we note that even very simple transformation design guidelines can limit the number of conflicts to manage and reduce the overhead of the proposed work-flow.

The contributions of the paper can be summarized as follows:

- C1 We introduce, in Section 3, a novel viewpoint on the integration of handwritten and generated code in MDD, inspired by the way in which concurrent changes to the code by multiple developers are handled. The core idea is to organize the MDD process according to a **model and code co-evolution** work-flow, which preserves the code changes (delta) between two consecutive generationsred. The availability of deltas enables the efficient comparison of the modifications introduced by the code generator, which recreates the code after a model update; and those made by the programmer, who modifies the generated code to implement non-modeled features. In this way, the code generator is treated as a **Virtual Developer**, who works together with human developers

[1] https://www.webratio.com [2] https://www.mendix.com [3] https://www.outsystems.com [4] https://www.zoho.com/creator

seamlessly. The Virtual Developer is a metaphor that underlines that the model and code co-evolution work-flow manages the code generated by MDD tools and by humans in a similar way.

C2 We compare, in Section 4, the model and code co-evolution work-flow with the use of protected areas and template-based transformations and show that it delivers superior flexibility w.r.t. protected areas, because it does not require the sharp separation between handwritten and generated code, and eliminates the meta-programming skills needed for template creation. The price for such benefits is the overhead of resolving conflicts. However, the availability of the incremental changes (deltas) of the human and virtual developer enables the use of simple VCS techniques to automate conflict resolution. The functionality of common VCSs can be used to identify collisions and either solve them automatically or point them out to the human developer for manual resolution.

C3 We study the practical viability of the model and code co-evolution work-flow, in Section 6, by comparing it to the template-based forward engineering process. To this end, we implement the work-flow on top of the popular Git[5] VCS (Section 5) and use it to develop two use-case applications in four ways: with the proposed work-flow and with a template-based process, in two MDD platforms: a prototyping tool (IFMLEdit.org [Bernaschina et al. 2017]) and an industrial tool (WebRatio). The comparison focuses on the amount of work required by the bottleneck activities: conflict resolution (for the model and code co-evolution work-flow) and template programming (for the template-based process). The assessment shows that, in IFMLEdit.org, the 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.

C4 As a collateral observation, in Section 6.2, we discuss how very simple guidelines for the design of code generators can reduce collisions. We evaluate their utility by re-implementing the code generator of IFMLEdit.org, so to incorporate the design guidelines, achieving a further reduction of the integration work.

The paper is organized as follows: Section 2 surveys the relevant literature. Section 3 introduces the model and code co-evolution work-flow (C1). Section 4 focuses on the management of conflicts between handwritten and generated code (C2). Section 5 describes the implementation of the proposed approach, obtained by integrating IFMLEdit.org and Git. To show the generality of the method, a second implementation integrates Git with a commercial tool (WebRatio) (C3). Section 6 evaluates the impact of the proposed work-flow in two use cases, with the template-based forward engineering process and the model and code co-evolution process (C3); it also assesses the effectiveness of the proposed transformation design guidelines (C4). Finally, Section 7 draws the conclusions and gives an outlook on future work.

2 RELATED WORK

The research problem tackled in this work is the integration between manually programmed and generated code, which arises due to the existence of non-modeled application features in MDD development processes.

The previous works can be classified in two families:

(1) Approaches that separate modeled and non-modeled features. The former are implemented by model-to-text transformations; the latter with handwritten code, integrated with the generated code with language-specific or language-agnostic mechanisms.

(2) Approaches that aim at promoting the manually programmed code to a “higher” status, mapping it either into a *template* or into a *model feature*. Such a promotion can be pursued with different techniques: templatization,

[5] https://git-scm.com

transformation inversion, and traceability links. After promotion, the part of the application that was originally coded manually can be re-generated automatically, with standard forward engineering.

In our approach the generated and manual parts of the application have equal status and their inconsistencies are resolved semi-automatically. Manual modifications are merged in the code-base seen by both the human programmers and by the MDD tools and automatically transported from one application version to the next.

2.1 Mechanisms for integrating handwritten and generated code

MDD methodologies acknowledge the reality that not all features of an application can be represented by the model, and the consequent necessity that non-modeled features must be coded manually. Therefore, the integration of handwritten and generated code has emerged as a primary issue in MDD. All the approaches share the assumption that the handwritten and generated code must be managed separately. Specifically, the handwritten code must be structured, and possibly exposed to the code generator, in such a way that the generator does not overwrite the manual modifications.

[Greifenberg et al. 2015] present six language-based and two language-independent integration mechanisms, which the designer of transformation rules can exploit to reduce collisions. [Kelly and Tolvanen 2008] also discuss pragmatic approaches, such as protected areas, inclusion of code in models, templates, and language- or framework-specific patterns. Protected areas are a simple mechanism to integrate MDD with manually programmed code; they define portions of the application for which the code is provided by the developer and is not overwritten by the code generator. For example, the industry standard transformation framework Acceleo[6] offers protected areas, which are preserved between executions of the model transformation. Also the work [Völter and Bettin 2004], which discusses MDD organizational practices, provides guidelines for improving the separation of handwritten and generated code.

Our approach does not require the strict separation of handwritten and generated code. Thanks to the proposed work-flow, when the generator recreates the code from the model, the changes since the last generation are compared with the manual changes made by developers; and VCS techniques are employed to identify conflicts and resolve collisions.

2.2 Transformation Evolution and Maintenance

Various approaches help transformation developers cope with manually programmed code and with change in the meta-models or in the architecture. Template-based code generation integrates examples of handwritten code (templates) into the model-to-text transformation [Stahl and Völter 2006]; the transformation produces the implementation code by iterating over the model elements and applying the relevant templates to selected model patterns. The need of evolving transformations can arise from changes in both the source meta-model and in the target technologies. The approaches to manage change in the meta-model are surveyed in [Hebig et al. 2017], which classifies the literature on the co-evolution of meta-models and models, including the techniques for propagating meta-model changes not only to the models but also to the transformations. An example of the latter case is [Garcia et al. 2013], which describes a tool that propagates semi-automatically changes of the input meta-model to the transformation that depends on it, aiding the developer to co-evolve the meta-model and the transformation. In [Hoisl and Sobernig 2015] transformation evolution is formalized and an approach to manage change after a meta-model update is discussed. The authors introduce goals and metrics to benchmark alternative methods for refactoring transformations after a meta-model change, with a perspective similar to the one adopted in this paper to quantitatively compare alternative ways to integrate manual updates into generated code. The approach of [Kövesdán et al. 2014] exploits dynamic template invocation to simplify the maintenance of

[6] https://www.eclipse.org/acceleo

transformations. [García et al. 2014] surveys the approaches to organize model transformations, analyzing the effects of moving rapidly evolving aspects outside the MDD work-flow into an abstraction layer managed manually. The work [Possatto and Lucrédio 2015] addresses template maintenance: the authors explore a mechanism for automatically updating templates after changes in the manually programmed code from which they derive. The method applies only to maintenance, because the initial definition of the templates entails the design of their logic, which is hard to automate.

The above mentioned approaches exploit features of the meta-models and of the transformation infrastructure, to either simplify transformation development or to support the promotion of manual code into templates.

Transformation maintenance approaches can help the integration of non-modeled features into the generated code, because they support the developer in updating either the transformation itself or the templates used by the transformation. However, they assume that the developer is familiar with transformation and template programming. Conversely, our technique does not require the programming of transformations and of templates, because it works only on the generated code. As a consequence, it is also more general, because it does not depend on the modeling and transformation languages.

2.3 Round trip engineering for model and code co-evolution

Round-trip engineering [Antkiewicz and Czarnecki 2006] is the capacity of synchronizing related software artifacts, such as models and code. Approaches fall in two classes: bi-directional transformations and trace-based techniques.

Bidirectional Transformations Bidirectionality aims at maintaining consistency among interdependent artifacts [Gibbons and Stevens 2018]. In MDD, bidirectional transformations support reversing the mapping from a model to another model [Hidaka et al. 2016]; when applied between a model and the code, they enable the reconstruction or enrichment of model features from source code. [Anjorin et al. 2010] use the Abstract Syntax Tree (AST) of the target language in a bidirectional transformation, defined via Triple Graph Grammar. The AST is enriched to support text chunks introduced during manual modifications and not managed by the transformation. [Greiner and Buchmann 2016] discusses the use of the QVT transformation language to implement the bidirectional transformation between UML class diagrams and Java code; the approach handles the structure and type information of Java programs, but not behavioral aspects, such as the method body. In general, addressing code-level changes by lifting them at the model level can reduce the complexity of modeling and transformation, at the cost of defining a specific parser for the target language and a reverse mapping from low level code to the high level concepts; this approach normally works well for a limited class of code-level constructs, most notably those representing the type and structural information of a program.

Traceability of model transformations In MDD, methods based on traceability links relate source model elements to corresponding target model elements [Santiago et al. 2012; Vara et al. 2014; Winkler and von Pilgrim 2010]. Traceability, applied to a model-to-text transformation, supports the analysis of the impact of a model update onto the generated code and the reconciliation between manually and automatically generated code. In [Ciccozzi et al. 2013] the authors integrate traceability links into a code generation framework, to monitor non-functional requirements. They exploit traceability links to enable the back-porting of the values of non functional properties, evaluated on the source code, to the modeling level, so that successive generations can be checked for the preservation of such properties. [Goldschmidt and Uhl 2013] present a tracing framework for change retainment, which helps tracing code-level modifications back to the model. The recent work [de Lange et al. 2018] pursues the goal of supporting collaborative MDD at both the modeling and coding level; a trace-based approach is exploited to detect modifications in the model and source code. After an update, a trace model is used to merge manual changes into generated code, so that the code-base reflects both model and manual code refinements. The approach requires an ad hoc template engine, producing the needed trace models, and limits the intervention of the programmers to fine-grained protected segments of the source code.

Bidirectional transformations and traceability systems can help developers to assess the impact of manual code modifications on the input model and to devise the proper strategy for merging handwritten and generated code; however, they are bound to the target languages and transformation frameworks. Supporting different languages or new features of a target language requires updating the transformation and tracking chains. The approach of this paper does not require the bidirectionality of the transformation nor ad hoc traceability frameworks and poses minimal requirements on the collaborative development process, transformation, input and output languages; these benefits come at the cost of manually resolving some of the conflicts between handwritten and generated code.

2.4 Software merging and conflict resolution

Software merging is the process of integrating multiple versions of a system, supporting developers in doing changes and removing inconsistencies. Merging is assisted by *Version Control Systems (VCS)*, which allow developers to work on local copies of the code, which are reconciled to produce a shared version. Merging has been studied for a long time [Mens 2002]. Techniques can be grouped into categories: *text-based* approaches consider a software artifact as a plain file and apply merging at the level of text lines; *syntactic* techniques also consider the syntax of the language and can avoid the production of unnecessary conflicts, e.g., those arising inside comments; *semantic* techniques consider the semantics of the language, which enables the detection of conflicts that a purely syntactic check would not identify; finally, *operation-based* techniques consider not only the artifacts but also the operations made by developers on them. With the advent of MDD, versioning and merging have been applied not only to the code, but also to models; the survey work [Altmanninger et al. 2009] offers a panorama of the VCS that have been applied for model versioning and of the outstanding research problems. [Cicchetti et al. 2013] exploits model differencing techniques to propagate the changes applied to a model to the artifacts depending on it, including those representing non-modeled features. The proposed approach requires specific transformations for all the possible types of model changes; for the data model, it can migrate the database schema and instance after a change, but for the presentation layer can only compute hints for the manual propagation of change. Commercial MDD tools, including Mendix, WebRatio, and Outsystems, offer functions for model versioning, visualization of model changes, and conflict resolution.

For the evaluation of the proposed approach, which detects the conflicts between handwritten and generated code, we have integrated two MDD environments with the Git VCS, which uses text-based, line-level merging. Even with such a basic technique, promising results are achieved. Since our approach is orthogonal to conflict resolution methods, it can be integrated with more sophisticated algorithms, such as fine grained/language specific resolution techniques [Birken 2014], leaving manual intervention as a fall-back. As future work, we plan to address conflict resolution also in the modeling phase, so to provide a unique environment for managing change at the model and at the code level.

3 APPROACH

The approach to the integration of handwritten and generated code proposed in this paper is rooted in two main ideas. The first one is letting the code generator and the human developers update the same code concurrently. The second is identifying and resolving the possible inconsistencies that these concurrent updates may produce. The method is inspired by the way in which VCSs handle concurrency among human developers, but with an important distinction due to the different behavior of the code generator and of human developers.

The code generator is invoked after a model change and always reproduces the whole code of the application or a large part thereof, thus potentially overriding *all* the previous manual modifications of the generated code. By keeping a detailed record of the contributions of the code generator the changes that it introduces at each invocation can be computed. The isolation of such changes enables the analysis to focus only on the last round of manual modification and code re-generation, and makes the problem tractable and resolvable with very simple VCS techniques.

Before proceeding, we introduce the concepts used in the rest of the paper.

- **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[7].
- **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 artifacts they contain is the same.
- **Difference**: the difference $R_{i,j} - R_{m,n}$ is the set of changes that need to be applied to $R_{m,n}$ to produce $R_{i,j}$.
- **Delta**: the delta induced by $R_{i,j}$ ($\Delta_{i,j} = R_{i,j} - R_{i,j-1}$) is the difference of $R_{i,j}$ 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. Conversely, 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 resulting from a manual change as $R^M_{i,j}$ (**manual revision**), to one containing generated artifacts as $R^G_{i,j}$ (**generated revision**), and to one resulting from conflict resolution as $R^R_{i,j}$ (**resolution revision**).

Fig. 1 illustrates the framework supporting the model and code co-evolution work-flow, the relevant roles and operations. The Modeler and the Programmer(s) work in their development environments: the former edits the model

[7] The generic term "code-base" maps to the notion of branch in specific systems, such as Git.

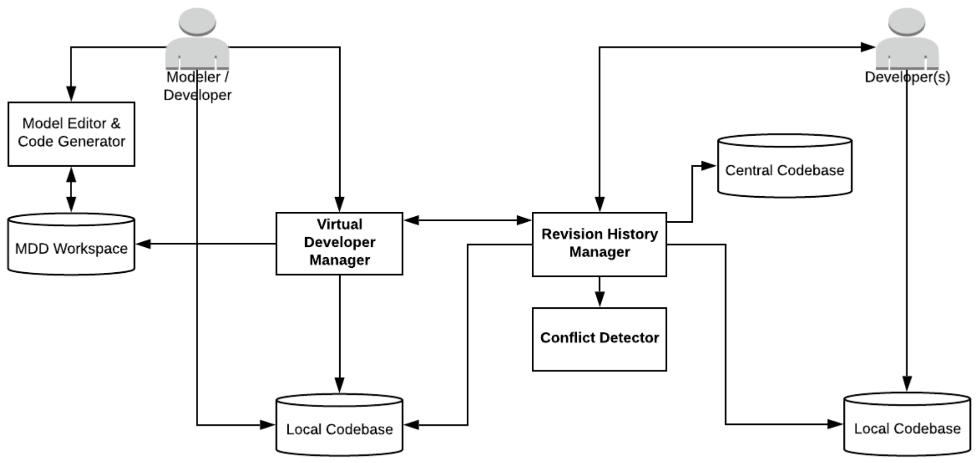


Fig. 1. Architecture supporting the model and code co-evolution work-flow

and generates the code; the latter can modify the generated code to implement non-modeled features. When a conflict arises between the handwritten and the automatically generated code, the Modeler (or a Programmer) is responsible of resolving the inconsistencies too. The local code bases contain the revisions produced by human developers and by the Code Generator, which is considered as a Virtual Developer; the central code base stores the shared version of the code resulting from the reconciliation of local updates. All submitted revisions, originated by human developers and by the Code Generator, are under the control of a *Revision History Manager*, which handles both the central shared code-base and the local copies produced by the Programmers and by the Code Generator. The dialogue between the Code Generator and the Revision History Manager is mediated by the *Virtual Developer Manager*, which extracts the generated code from the MDD work space and processes its submission according to the procedures described in Section 3.3. Thanks to this module, the interface of the Revision History Manager towards the Code Generator is the same as that towards human developers. At each submission, a work-flow is triggered: the submission is passed by the Revision History Manager to the *Conflict Detector*, which checks if a conflict is produced. If there is no conflict, the Revision History Manager simply stores the revision in the central code-base. If a conflict is detected, the *Conflict Detector* identifies the possible *collisions* (i.e., the parts of the code updated concurrently in an inconsistent manner); if these exist and the Conflict Detector has rules for resolving them automatically, it applies them; otherwise, the Revision History Manager notifies a human developer (the Modeler or a Programmer), who has to resolve them and produces a new revision in the local code base of the Code Generator; the Revision History Manager handles its submission and stores the novel consistent revision in the central code-base. Note that the Virtual Developer Manager acts as a proxy to the Modeler, who starts it after code generation and resumes it after an interactive conflict resolution session, to close the submission.

In Section 5 we show how the architecture of Fig. 1 can be implemented on top of a popular VCS.

In the rest of this Section, we analyze how parallel development is managed (Section 3.1), show similarities between parallel development and model and code co-evolution (Section 3.2), and apply parallel development methodologies to the co-evolution of model and code, by treating the MDD tool-chain as yet another developer in the team (Section 3.3).

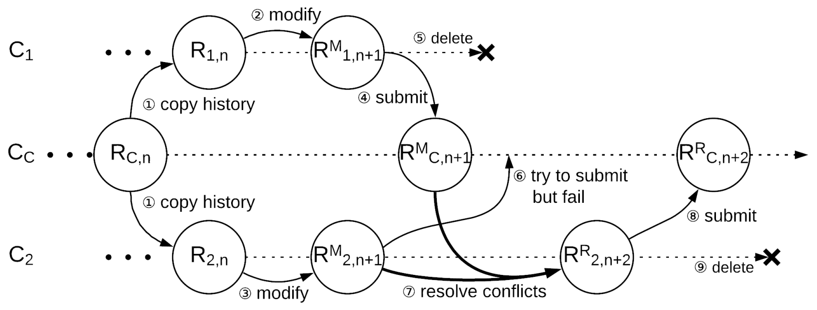


Fig. 2. Development with conflicts

3.1 Parallel & Distributed Development

In parallel development, different subjects introduce distinct new features into the code-base independently. When two developers D_1 and D_2 work in parallel on the same system, the work-flow may look like the following:

Work-flow: development with conflict

(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 $\forall_{j\in\{1..n\}} R_{1,j} = R_{C,j} = R_{2,j}$.
(2) D_1 introduces a new feature by applying changes to $R_{1,n}$, creating a new revision $R^M_{1,n+1}$.
(3) D_2 independently 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 central code-base generating a new revision $R^M_{1,n+1} \longrightarrow C_C = R^M_{C,n+1}$. $R^M_{C,n+1}$ 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 submits $R^M_{2,n+1}$ to the central code-base. The operation fails because $R^M_{2,n+1}$ creates a conflict with C_C.
(7) D_2 performs conflict resolution between $R^M_{2,n+1}$ and C_C: $R^M_{C,n+1} \longmapsto C_2$; this step generates a new local revision $R^R_{2,n+2}$, which integrates the feature locally developed by D_2 and the current state of the central code-base (which, in this case, comprises the feature developed and submitted by D_1).
(8) D_2 submits $R^R_{2,n+2}$ to the central code-base $R^R_{2,n+2} \longrightarrow C_C$; the submission now is accepted because $R^R_{2,n+2}$ is the result of the conflict resolution $R_{C,n+1} \longmapsto C_2$. This step produces a new shared revision $R^R_{C,n+2}$.
(9) D_2 deletes her local copy of the code-base C_2 returning to the initial state. Her work is safely stored in C_C.

After such a work-flow, the revision history shown in Fig. 2 is achieved.

3.2 Model and Code Co-Evolution

By allowing manual editing on the generated code, it is possible to introduce conflicts with code generated from a new model version. A naïve approach to the resolution of such conflicts is the one realized by the following work-flow:

Work-flow: development with naïve conflict management

(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 performs a submission to the central code-base $R^M_{1,2} \rightarrow C_C$. The new revision $R^M_{C,2}$ 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 revision $R^G_{1,1}$ (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 central 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, generated from the current model, and is not created by evolving a preceding shared revision stored 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 ($R^M_{C,2} \mapsto C_1 = R^R_{1,2}$). Conflict resolution can be done manually, by merging the manual modifications to the newly generated code, or supported by tools, as discussed in Section 4.
(10) She submits the new revision to the central 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 Fig. 3 is achieved.

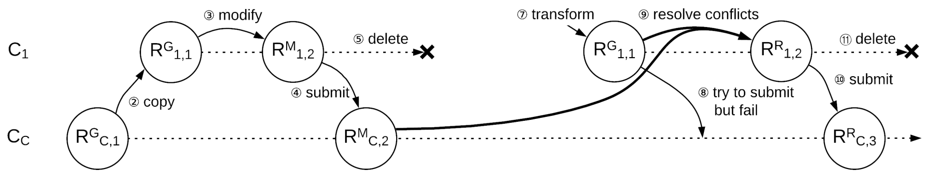


Fig. 3. Development with naïve conflict management

3.3 The Virtual Developer

The developer D_2 in Section 3.1 works on an outdated version of the code-base (see step 3 and 6 in Fig. 2), which requires the conflict to be solved. The model-to-text transformation in Section 3.2 generates a revision $R^G_{1,1}$ (see step 7 and 8 in Fig. 3) in conflict with the central code-base due to the previous manual update of the code. These two scenarios differ w.r.t. the agent whose changes generate the conflict, but the evolution of the textual artifact follows a similar path in both cases. In the work-flow of Fig. 3, after a model update, the newly generated revision $R^G_{1,1}$ (created at step 7) is used to initialize the local code-base C_1 afresh; this is due to the fact that the model-to-text transformation does not evolve the code-base incrementally, as real developers do, but regenerates the whole textual artifact. At every generation, it would be as if an update on the entire artifact is checked in to the repository regardless of the past history of the code. Submitting a generated revision without precautions would “forget” the conflict resolution steps done in precedence and make the same sources of conflict arise again at each code generation. But if the human developer has modified the code generated from a model element E and the subsequent generation applies to a model in which E has not been updated, then the manual modification is still valid and should not be considered as a source of conflict. To make the behavior of the code generator more similar to that of the human developer, redundant information can be inserted in the code-base history, simulating the incremental production of code by the model transformation. Fig. 4 shows the result of such an approach: the work-flow is equivalent to the one of Fig. 3, but with redundant information; 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 generated revision $R^G_{1,1} = R^G_{C,1}$, before overwriting it with the new generated revision ($R^G_{1,2}$ in Fig. 4). The initialization of the local code-base with the latest generated revision before code generation makes it possible to compute the delta $\Delta^G_{1,2}$ induced by the generation. Even if the transformation overwrites the entire artifact, $\Delta^G_{1,2}$ singles out exactly the *incremental* changes that the transformation induces over the previously generated revision $R^G_{1,1}$ after the model update. This permits the conflict resolution to focus only on the collisions that derive from the manually modified code of a model element for which the current round of generation has produced new code.

Besides the difference in the initialization of the code-base, the manual and automatic generation of the artifacts now follow the same path, as visible by comparing the work-flow of Fig. 2 and Fig. 4. Revisions $R^M_{2,n+1}$, of Fig. 2, and $R^G_{1,2}$, of Fig. 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 ($R_{2,n}$ 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 the shared code-base 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 single Virtual developer can work in parallel, in a work-flow such as the following:

Work-flow: the Virtual Developer

(1) C_C contains n revisions, the latest revision $R^G_{C,n}$ is 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 central code-base generating a new revision $R^M_{H,n+1} \longrightarrow C_C = R^M_{C,n+1}$. 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 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.

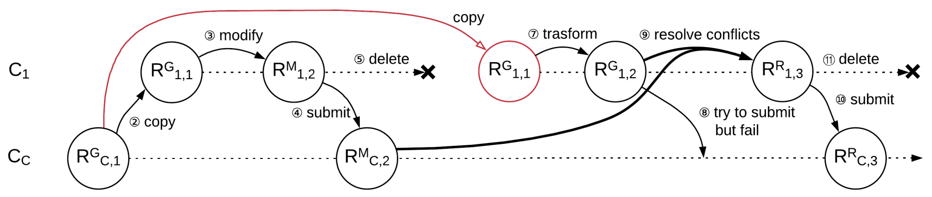


Fig. 4. Work-flow with redundant information to compute the delta after model update and code generation

(9) D_V tries to submit $R^G_{V,n+1}$ to the central 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 4.

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

(12) D_V submits the conflict resolution $R^R_{V,n+2}$ to the central code-base, generating a new shared revision $R^R_{V,n+2} \to C_C = R^R_{C,n+3}$. 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 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.

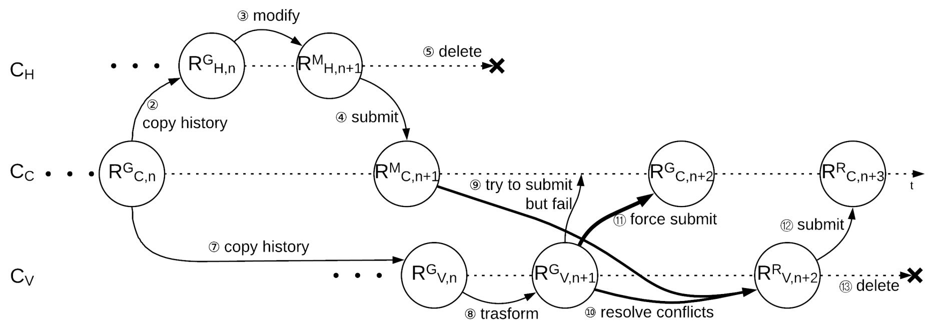


Fig. 5. Work-flow with the Virtual Developer; the revision history contains redundant information.

Fig. 5 shows the history that results from considering the model-to-text transformation as a Virtual Developer. The central code-base contains a twofold sequence of revisions: automatically generated ($R^G_{C,j}$) and conflict-resolved ($R^R_{C,j+1}$). 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 $R^G_{C,j}$.

Note that the requirement that the central code-base stores both generated and conflict-resolved revisions is not mandatory. An alternative organization can be as follows. The central code-base C_C keeps only the "valid" revisions (the first one is $R^G_{C,1}$ and the last one is $R^M_{C,n}$ if the latest change was a manual modification or $R^R_{C,n}$, if the latest change was a regeneration). The local code-base of the Virtual Developer C_V maintains all the generated revisions. When the human developer needs to apply manual changes to the generated code, she copies the last version from the central code-base into a *transient* local code-base C_t, applies the manual modification, submits the resulting version to the central code-base and then deletes the transient code-base. When the code is regenerated by the Virtual Developer, a new revision is submitted to C_V, a *transient* code-base C_t is also created, the delta of the code generator is computed in the transient code-base, a resolution revision is submitted to the central code-base, and finally, deleted. This organization performs the reconciliation within a transient version C_t, without interfering in the histories of C_C or C_V, at the price of more complexity in the implementation of the Virtual Developer Manager, which must reconstruct the delta of the code generator from the persistent and the transient code-base.

In Section 4.4 we introduce a complete example of the Virtual developer work-flow and provide in the Appendix a graphical representation of the work-flow and a legenda of the relevant revisions and deltas.

4 AUTOMATING CONFLICT RESOLUTION

Purely manual approaches to the resolution of conflicts are impractical, as they cannot scale with the size of the code-base. In this Section, first we show how the proposed work-flow can accommodate industry-standard approaches such as template-based forward engineering and protected areas; then, we show how the revision deltas can be exploited to relax the constraints of code generation with template-based transformations and with protected areas.

4.1 Template-based forward engineering

Template-based forward engineering incorporates customizations of the output in templates, which the generator applies to produce the code. After each model change, the entire artifact is regenerated and used to replace the previous version: the revision produced by the conflict resolution $R^M_{C,n+1} \mapsto C_V = R^R_{V,n+2}$ is always equivalent to $R^G_{V,n+1}$, i.e., manual changes introduced in the code-base C_C are ignored. While conflict resolution can be fully automated and no human intervention is required, the price to pay is the creation and maintenance of templates, a task that requires programming 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.

4.2 Protected areas

With protected areas, manual changes are confined to special portions of the project. Conflict resolution can be automated, by using the content of revision $R^G_{V,n+1}$ as the base to build $R^R_{V,n+2}$. Each area is identified by a unique signature, preserved between executions of the transformation. If an area is present in both $R^M_{C,n+1}$ and $R^G_{V,n+1}$, its content is extracted from the central code-base $R^M_{C,n+1}$ and inserted into the matching protected area of the local revision $R^R_{V,n+2}$, possibly replacing the generated code. If an area is present in $R^M_{C,n+1}$, but not in $R^G_{V,n+1}$, e.g., due to the deletion of the model element that contained it, its content is not reinstalled in $R^G_{V,n+2}$. With protected areas, no back-porting of features into templates is needed; but the handwritten code must be separable from the rest of the code, an assumption that is hardly verified in web and mobile applications, where the presentation is entangled with the structure of the front-end and new non-functional requirements are frequently introduced during the project lifetime.

4.3 Exploiting deltas

The approach introduced in this paper can be used in conjunction with both template-based code generation and protected areas, but with a different relevance. If a (pure) template-based approach is used, then our approach is applicable but becomes rather useless, because the assumption of using templates is that the developer does not modify the generated code and thus there are no conflicts. If code generation with protected areas is used, then conflicts do arise; however, they have a special, constrained, format (they occur only in protected regions) and thus conflict resolution can be fully automated, as explained in Section 4.2.

Thus, the focus of this paper is to relax the assumptions of templates and protected areas, to support the resolution of conflicts between handwritten and generated code in more situations. By exploiting the redundant information stored in the revision history, it is possible to limit human intervention to those cases in which a conflict resolution strategy cannot be inferred, which are less than 30%. 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 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 $R^G_{V,n}$ (see step 7 in Fig. 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. It is precisely in those two deltas that the interference between the automatically generated and the manually written code 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 position 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**: an undirected graph (henceforth denoted by U) in which vertexes are updates and edges, if present, denote the collision between them.
- **Collision Group**: (henceforth denoted by 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** (henceforth denoted by U_F); the updates in a collision-free graph can be applied independently.
- **Collision Resolution**: a procedure that takes in input a collision graph and produces a collision free graph.

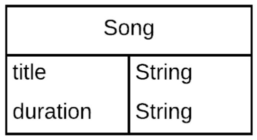
Conflict resolution can be achieved with the following steps:

(1) $\Delta^M_{C,n+1}$ is decomposed into its constituent (collision-free) updates U^M_F.
(2) $\Delta^G_{V,n+1}$ is decomposed into its constituent (collision-free) updates U^G_F.
(3) The graph $U_C = U^M_F \oplus U^G_F$ is formed; the binary $\oplus$ operator builds an update graph by taking the union of the vertices of the two input graphs and computing the edges according to the definition of collision given above; therefore, U_C can contain collisions.
(4) If U_C contains no collision groups, the updates to apply U are defined as the vertices of $U = U_C$. Otherwise, collision resolution is applied, so to create a new collision-free graph U_F and the updates to apply U are defined as the vertices of $U = U_F$.
(5) The updates in U, which are guaranteed non to collide, are applied to $R^G_{V,n+1}$ generating $R^R_{V,n+2}$.

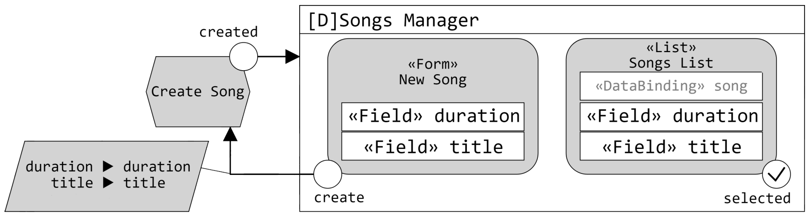
The above mentioned steps are supported in all industry standard VCSs: they automatically identify the set U, by decomposing $\Delta^M_{C,n+1}$ and $\Delta^G_{V,n+1}$, automatically produce $R^R_{V,n+2}$, if no collision groups are identified, or support the developer during collision resolution, if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

4.4 An illustrative example

We now exemplify the work-flow by means of a toy application for managing the content of a music repository. The example implements a simple content management functionality, whereby the user can create a new song, display the



(a) Domain Model



(b) IFML Model

Fig. 6. First version of the IFML model of the Song Manager

list of songs and delete an existing song. Three versions of the application are specified with IFML, which permits the expression of the views, of their content, of the events triggered by the user's interaction and the actions executed to handle such occurrences. In the Appendix, we provide a graphical representation of the work-flow and a legenda of the relevant revisions and deltas.

Step 1: initial model. The development starts with the IFML model of Fig 6: the application consists of a single page (*Songs Manager*), which displays a list of songs (denoted by the *Songs List* ViewComponent) and a form (denoted by the *New Song* ViewComponent); the former displays the title and duration of each song in the list, the latter comprises two fields to input the title and duration of a new song. The *New Song* form ViewComponent is associated with the *create* event, which triggers the *Create Song* action; the NavigationFlow from the form to the action is associated with two parameters (duration and title), which denotes that the field values entered by the user are passed as input parameters to the create operation. The IFML model assumes a Domain Model consisting of a single entity *Song*, which is used in the data binding of the *Songs List* ViewComponent.

Step 2: first code generation. The IFMLEdit.org code generator is invoked on the model of Fig. 6 to produce the first generated revision stored in the central code-base: $R^G_{C,1}$. *Songs List* is generated as an HTML view for the Knockout JavaScript library[8], with a default presentation.

```html
<h3>Songs List</h3>
<table class="table table-hover table-condensed">
    <thead>
        <tr>
            <th>#</th>
            <th>duration</th>
            <th>title</th>
        </tr>
    </thead>
    <tbody data-bind="foreach: items">
        <tr data-bind="click: $parent.select">
            <td data-bind="text: id"></td>
            <td data-bind="text: $data['duration']" style="white-space: pre-wrap;"></td>
            <td data-bind="text: $data['title']" style="white-space: pre-wrap;"></td>
        </tr>
    </tbody>
</table>
```

The *Create Song* action is generated as a JavaScript function with a dummy body.

[8] https://knockoutjs.com/

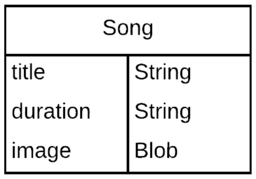
```
function Action() { // add "options" parameters if needed
    // TODO: Global Initialization
    /*
    you code here
    */
}
Action.prototype.run = function (parameters, solve) { // add "onCancel" parameters if needed
    // Parameters:
    // parameters['duration']
    // parameters['title']
    // TODO: Execution
    /*
    your code here
    */
    // DEFAULT TERMINATION: CAN BE REPLACED (BEGIN)
    $.notify({message: 'Create Song'}, {allow_dismiss: true, type: 'success'});
    solve({
        event: 'create-song-created', // created
        data: {
        }
    });
    // DEFAULT TERMINATION: CAN BE REPLACED (END)
};
```

The data access layer for the *Song* entity is generated with a default implementation, too. We omit it, for brevity.

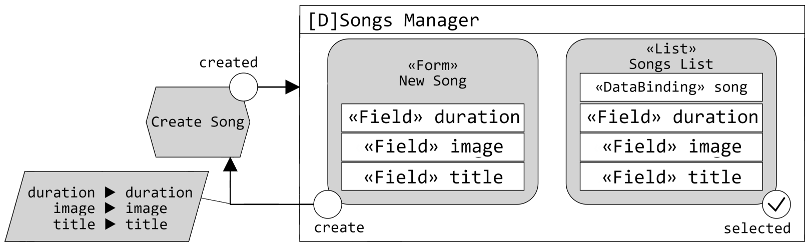
Step 3: first manual modification. The generated code is changed by the programmer to remove the internal id from the list view and to implement song creation by invoking the data access layer. The manual updates are submitted to the central code-base, which now contains the sequence: $R^G_{C,1}$, $R^M_{C,2}$. The delta $\Delta^M_{C,2} = R^M_{C,2}$ - $R^G_{C,1}$ is shown next. In the code that follows, a + at the beginning of a line means that it has been inserted, a - that it has been deleted.

```
 <table class="table table-hover table-condensed">
     <thead>
         <tr>
-            <th>#</th>
             <th>duration</th>
             <th>title</th>
         </tr>
     </thead>
     <tbody data-bind="foreach: items">
         <tr data-bind="click: $parent.select">
-            <td data-bind="text: id"></td>
             <td data-bind="text: $data['duration']" style="white-space: pre-wrap;"></td>
             <td data-bind="text: $data['title']" style="white-space: pre-wrap;"></td>
         </tr>

-function Action() { // add "options" parameters if needed
-    // TODO: Global Initialization
-    /*
-    your code here
-    */
+function Action(options) { // add "options" parameters if needed
+    this.collection = options.repositories.song;
 }
 Action.prototype.run = function (parameters, solve) { // add "onCancel" parameters if needed
-    // Parameters:
```



(a) Domain Model



(b) IFML Model

Fig. 7. Model 2: adding an image to songs

```
-    // parameters['duration']
-    // parameters['title']
--    // TODO: Execution
-    /*
-    your code here
-    */
-    // DEFAULT TERMINATION: CAN BE REPLACED (BEGIN)
-    $.notify({message: 'Create Song'}, {allow_dismiss: true, type: 'success'});
-    solve({
-        event: 'create-song-created', // created
-        data: {
-        }
+    this.collection.create({
+        duration: parameters.duration,
+        title: parameters.title,
+    }).then(function () {
+        solve({
+            event: 'create-song-created', // created
+            data: {}
+        });
     });
-    // DEFAULT TERMINATION: CAN BE REPLACED (END)
 };
```

The implementation of the data access layer is also extended by adding the code to create a new song.

Step 4: first update to the model. The application is extended by adding an *image* field to the *Song* entity, which requires adding a field to the form, an attribute to the list, and a parameter to the input of the action. Code generation produces the second generated revision $R^G_{V,2}$; this is submitted to the central code-base for conflict checking.

In $R^G_{V,2}$, the *Songs List* view is extended with the image field. In the following we show the $\Delta^G_{V,2}$, i.e., the difference *with respect to the previous generation of the code* ($R^G_{V,1}$). It is such a delta that will allow the Conflict Detector to focus the analysis only on the difference introduced by the last run of the code generator. Therefore, the intervening manual modifications (e.g., the removal of the id attribute from the song list) are ignored, and thus overridden, by the code generator.

```
    <thead>
        <tr>
            <th>#</th>
            <th>duration</th>
+           <th>image</th>
            <th>title</th>
        </tr>
    </thead>
    <tbody data-bind="foreach: items">
        <tr data-bind="click: $parent.select">
            <td data-bind="text: id"></td>
            <td data-bind="text: $data['duration']" style="white-space: pre-wrap;"></td>
+           <td data-bind="text: $data['image']" style="white-space: pre-wrap;"></td>
            <td data-bind="text: $data['title']" style="white-space: pre-wrap;"></td>
        </tr>
    </tbody>
```

In $\Delta^G_{V,2}$, the *Create Song* action dummy implementation is extended with the extra parameter. Also in this case, the code generator overwrites the manual implementation of the action body.

```
 Action.prototype.run = function (parameters, solve) { // add "onCancel" parameters if needed
     // Parameters:
     // parameters['duration']
+    // parameters['image']
     // parameters['title']
     // TODO: Execution
```

The data access layer for the *Song* entity is the same of the $R^G_{V,1}$.

Step 5: Submission with conflict and collision. Submitting the generated revision produces a conflict, because $R^G_{V,2}$ is not derived from $R^M_{C,2}$. Therefore conflict resolution must be performed. The part of $\Delta^G_{V,2}$ relative to *Songs list* does not create collisions: the code generator has only added the HTML `td` element for the image field and has not changed the rest of the code; therefore, the manual changes done at step 3 (the deletion of the column header and cell with the song id) do not collide with $\Delta^G_{V,2}$ and are preserved.

Conversely, the part of $\Delta^G_{V,2}$ relative to the *Create Song* action introduces collisions, because $\Delta^G_{V,2}$ and $\Delta^M_{C,2} = (R^M_{C,2} - R^G_{C,1})$ contain changes to the same lines. The human intervention resolves the collisions and produces the first resolved revision $R^R_{V,3}$. Then, both the last generated revision and the resolved revision are stored in the central code-base, which contains the sequence: $R^G_{C,1}, R^M_{C,2}, R^G_{C,3}, R^R_{C,4}$. The following shows a fragment of $\Delta^R_{V,3} = (R^R_{V,3} - R^G_{V,2})$, which is equivalent to $\Delta^R_{C,4} = (R^R_{C,4} - R^G_{C,3})$ after the submission to C_C.

```
-function Action() { // add "options" parameters if needed
-    // TODO: Global Initialization
-    /*
-    your code here
-    */
+function Action(options) { // add "options" parameters if needed
+    this.collection = options.repositories.song;
 }
 Action.prototype.run = function (parameters, solve) { // add "onCancel" parameters if needed
-    // Parameters:
-    // parameters['duration']
-    // parameters['image']
-    // parameters['title']
-
-    // TODO: Execution
-    /*
-    your code here
-    */
-    // DEFAULT TERMINATION: CAN BE REPLACED (BEGIN)
-    $.notify({message: 'Create Song'}, {allow_dismiss: true, type: 'success'});
-    solve({
-        event: 'create-song-created', // created
-        data: {
-        }
+    this.collection.create({
+        duration: parameters.duration,
+        image: parameters.image,
+        title: parameters.title,
+    }).then(function () {
+        solve({
+            event: 'create-song-created', // created
+            data: {}
+        });
     });
-    // DEFAULT TERMINATION: CAN BE REPLACED (END)
 };
```

The generator did not alter the data access layer for the *Song* entity, so the manual changes of $R^M_{C,2}$ are preserved.

Step 6: manual modification of the generated code to show the song image. The human developer initializes a new code-base with the last revision from the central code-base ($R^R_{C,4}$) and produces a new manual revision $R^M_{H,5}$; this modifies the implementation of *Songs List*, to move the position of the image column in the HTML table and to render the image with the HTML `img` tag. The following shows the delta of the human developer ($\Delta^M_{H,5} = R^M_{H,5} - R^R_{H,4}$).

```
     <thead>
         <tr>
-            <th>duration</th>
             <th>image</th>
+            <th>duration</th>
             <th>title</th>
         </tr>
     </thead>
     <tbody data-bind="foreach: items">
         <tr data-bind="click: $parent.select">
+            <td>
+                <img data-bind="attr: {src: $data['image']}" width="300px" class="img-thumbnail">
+            </td>
             <td data-bind="text: $data['duration']" style="white-space: pre-wrap;"></td>
-            <td data-bind="text: $data['image']" style="white-space: pre-wrap;"></td>
             <td data-bind="text: $data['title']" style="white-space: pre-wrap;"></td>
         </tr>
     </tbody>
```

The manual revision is submitted to the central code-base and produces $R^M_{C,5}$.

Step 7: adding functionality to delete songs to the model. The modeler adds an event to *Songs list* and a *Delete song* action triggered by it; the id of the selected song is passed to the action as a parameter of the NavigationFlow connecting the event to the action.

In the regenerated revision ($R^G_{V,4}$), the implementation of the *Songs List* ViewComponent is extended with an extra table column hosting the links to trigger the song delete action.

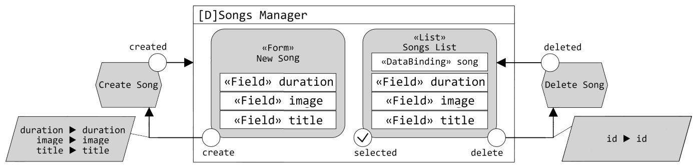


Fig. 8. Model 3: adding functionality to delete a song

```
                <th>duration</th>
                <th>image</th>
                <th>title</th>
+               <th>actions</th>
            </tr>
        </thead>
        <tbody data-bind="foreach: items">
            <tr data-bind="click: $parent.select">
                <td data-bind="text: $data['duration']" style="white-space: pre-wrap;"></td>
                <td data-bind="text: $data['image']" style="white-space: pre-wrap;"></td>
                <td data-bind="text: $data['title']" style="white-space: pre-wrap;"></td>
+               <td>
+                   <a class="col-xs-2 btn btn-primary" data-bind="click: $parent.trigger.bind($data, 'songs-list-delete'),
      clickBubble: false">delete</a>
+               </td>
            </tr>
```

Furthermore the code of the *Delete Song* action is generated with a dummy implementation. The generated action implementation is similar to that of the *Create Song* action, shown before; thus, we omit it for brevity.

Step 8: conflict resolution without collision . When the generated revision $R^G_{V,4}$ is submitted, a conflict is raised, but the new generated code of *Songs List* does not introduce a collision because only new lines (the th and td of the new *actions* column) are added to code not affected by the intervention of the human developer; thus the previous manual updates are preserved and the new generated code is merged. $\Delta^R_{V,5}$ contains the following updates for *Song list*.

```
        <thead>
            <tr>
-               <th>#</th>
-               <th>duration</th>
                <th>image</th>
+               <th>duration</th>
                <th>title</th>
                <th>actions</th>
            </tr>
        </thead>
        <tbody data-bind="foreach: items">
            <tr data-bind="click: $parent.select">
-               <td data-bind="text: id"></td>
+               <td>
+                   <img data-bind="attr: {src: $data['image']}" width="300px" class="img-thumbnail">
+               </td>
                <td data-bind="text: $data['duration']" style="white-space: pre-wrap;"></td>
-               <td data-bind="text: $data['image']" style="white-space: pre-wrap;"></td>
                <td data-bind="text: $data['title']" style="white-space: pre-wrap;"></td>
            <td>
                <a class="col-xs-2 btn btn-primary" data-bind="click: $parent.trigger.bind($data, 'songs-list-delete'),
                     clickBubble: false">delete</a>
```

The implementation of the *Create Song* action is not altered by the code generator, so no collision arises and the previous manual updates are preserved too.

Step 9: manual modification of the generated code to implement the Delete Song action. As a last step, the programmer now modifies the generated code to implement the *Delete Song* action, by invoking the data access layer. She makes a local copy of the code-base up to $R^R_{C,7}$, updates the code and submits it to the central code-base, creating the last revision $R^M_{C,8}$. The following shows the relevant part of the $\Delta^M_{H,8} = R^M_{H,8}$ - $R^R_{H,7}$.

```
-function Action() { // add "options" parameters if needed
-    // TODO: Global Initialization
-    /*
-    your code here
-    */
+function Action(options) { // add "options" parameters if needed
+    this.collection = options.repositories.song;
 }
 Action.prototype.run = function (parameters, solve) { // add "onCancel" parameters if needed
-    // Parameters:
-    // parameters['id']
-    // TODO: Execution
-    /*
-    your code here
-    */
-    // DEFAULT TERMINATION: CAN BE REPLACED (BEGIN)
-    $.notify({message: 'Delete Song'}, {allow_dismiss: true, type: 'success'});
-    solve({
-        event: 'delete-song-deleted', // deleted
-        data: {
-        }
+    this.collection.deleteById(parameters.id).then(function () {
+        solve({
+            event: 'delete-song-deleted', // deleted
+            data: {}
+        });
     });
-    // DEFAULT TERMINATION: CAN BE REPLACED (END)
 };
```

For brevity, we omit showing the extension of the data access layer for the *Song* entity. In the Appendix, we provide a graphical representation of the work-flow and a legenda of the relevant revisions and deltas.

5 IMPLEMENTATION

The architecture of Fig. 1 has been implemented with two different environments: WebRatio, a product for the development of Web and mobile application, and IFMLEdit.org [Bernaschina et al. 2017], an open-source environment for the rapid prototyping of Web and mobile applications. IFMLEdit.org permits the replacement of the code generator; thus it has been used for evaluating the proposed transformation design guidelines.

WebRatio is a commercial tool for the development of Web and mobile applications. The user interaction is defined via IFML, the domain model via UML Class Diagrams, and the business logic via a proprietary Action Definition

Language. The tool enables the generation of the full code of the application. Details that cannot be modeled directly (e.g., UI look & feel, application specific operations that implement IFML abstract Actions) are incorporated using a template-based approach. Templates are programmed with the Groovy scripting language; an extension of IFML allows the developer to tag each IFML element with the custom template to use for code generation.

IFMLEdit.org [Bernaschina et al. 2017] is an online environment for the specification of IFML models and the generation of prototypes of Web and mobile applications. 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 [Bernaschina 2017] transformation framework, which allows the developer to specify model transformations with a rule-based extension of JavaScript. ALMOsT.js supports the construction of template-based code generators, via the integration of templating languages, such as EJS [9]. 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[10] application.

VCS integration

The Revision History Manager and Conflict Detector modules of Fig. 1 have been developed by reusing the functionality of an existing VCS (Git), whereas the Virtual Developer Manager has been developed from scratch.

The Virtual Developer Manager has been realized as a middleware system, called ALMOsT-Git[11], which maps the operations required to handle a submission of the Code Generator into calls to the Revision History Manager interface. The functionality of the Revision History Manager and of the Conflict Detector has been realized by mapping their operations onto 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.
- 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 *pull* operation in Git.

ALMOsT-Git, given as input the location of the central repository and of the latest generated artifact, implements the work-flow of Section 3. It stops in case of collisions, highlighting the affected lines in the source code and asking the developer to solve conflicts. After resolution, she restarts the tool, which resumes and completes the work-flow.

5.1 Case studies

The model and text co-evolution work-flow and the collision prevention guidelines, which will be detailed on Section 6.2, have been evaluated in two use-cases. Both applications are sufficiently limited in size to be fully described, but feature non trivial requirements: access to remote resources, use of the multimedia and hardware capabilities of the device, and a customized presentation.

5.1.1 A Quiz Game. The application is a mobile phone Quiz Game, designed to extend the play of a real card game. It requires access to the camera, to scan QR codes on cards, and the network, to download questions and answers.

[9] https://ejs.co [10] https://cordova.apache.org [11] https://npmjs.com/package/almost-git

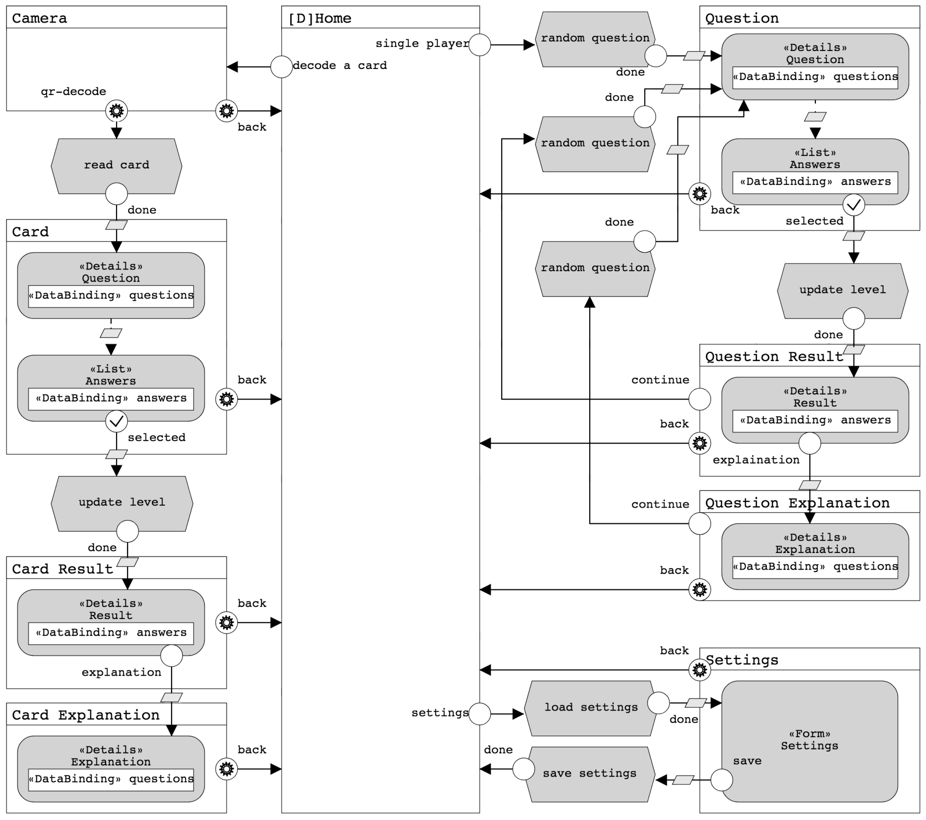


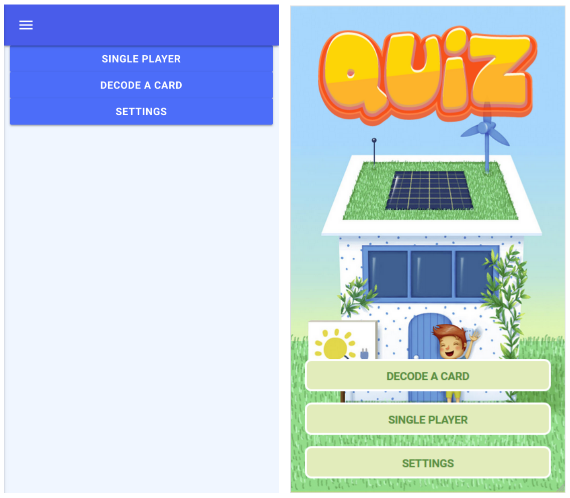
Fig. 9. Quiz Game: IFML model

The development, tracked in a public repository[12], followed four sprints:

(1) **Proof of Concept**. It allows the user to scan a QR code on a special card of the real game. After scanning, the game proposes a question on a subject related to the card (water and energy conservation). Once the user answers, the application shows the correct option and returns to the initial state.
(2) **Styling and Explanation**. A custom presentation is applied to the user interface, to give the application a professional look&feel. The application allows the user to see an explanation about the correct answer.
(3) **Secondary Game Mechanics**. A new command allows the user to skip the QR scanning and play in standalone mode, receiving a series of question of increasing difficulty.
(4) **Internationalization**. The possibility to select the language is introduced.

Fig. 9 shows the final IFML model. The application front end consists of seven screens. The *Home* page is void of content and supports only the activation of events: *decode a card*, *single player*, and *settings*. The triggering of the *decode a card* event opens the *Camera* page, which denotes the activation of the device camera to scan a QR code on the card; from such a page, the *back* system event brings back to the *Home* page and the *qr-decode* system event leads to the *Card* page. The *Card* page contains the *Question* component, which displays the text of the question associated with the scanned QR-code, and the *Answers* component, which displays the list of possible answers. The *selected* event denotes the choice

[12] http://github.com/emanuele-falzone/almostjs-git-demo-game



(a) Default (b) Custom

Fig. 10. Quiz Game: from prototype to final product

of an answer from the list and triggers the *update level* action, which checks the correctness of the answer, possibly changes the difficulty level, and opens the *Card result* page. The *Card result* page contains the *Result* component, which displays a message based on the outcome of the check. The *single player* event of the *Home* page fires the *random question* action, which selects a question randomly and opens the *Question* page. From the *Question* page, a sequence of events and pages supports the answering of questions, with an interaction similar to that triggered by the *decode a card* event, with the difference that a *continue* event associated with the *Explanation* page allows the player to iterate over a series of random questions. Finally, the *settings* event in the *Home* page fires the *load settings* action, which fills the form in the *Settings* page with the current values of the game settings (e.g., preferred language). From there, the *save* event fires the *save settings* action and returns to the *Home* page.

Fig. 10 shows the difference between the default UI generated by IFMLEdit.org and the final result. This application has been released as part of the card game Funergy[13].

5.1.2 Media Player. The second use-case application is a web-based Media Player, requiring interaction with the media capabilities of the device and the access to remote resources over the network.

The development, tracked on a public repository[14], followed 4 steps:

(1) **Proof of Concept**. The application loads a list of songs, allows the user to select the song to play, and to pause/restart it.
(2) **Styling**. A custom presentation style is applied and a system event is added to catch the end of a song.
(3) **Songs Filtering**. A filter is added, whereby the user can filter the song list by author.
(4) **Song Cover**. The interface is enhanced by showing the cover of the song currently playing.

The final IFML model for the application is visible in Fig. 11. The front end of the application comprises a principal page (*Application*), which displays two sub-pages (*Stopped* and *Playing*). The sub-pages are displayed in alternative, as

[13] http://play.google.com/store/apps/details?id=com.eu.funergy [14] http://github.com/emanuele-falzone/almostjs-git-media-player

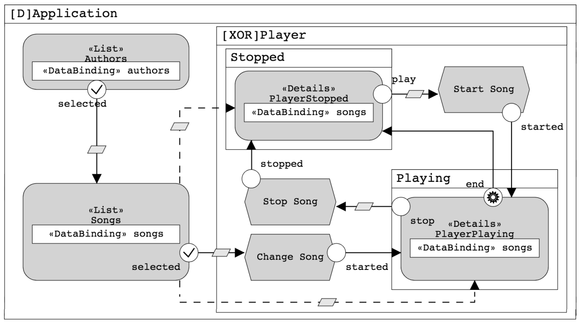
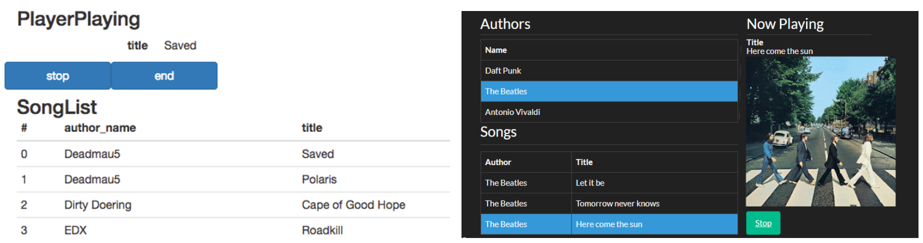


Fig. 11. Media Player: IFML model

denoted by the *Player* XOR ViewContainer that encloses them. The *Application* page contains the *Authors* component, which displays the list of available authors, and the *Songs* component, which displays the list of available songs of the author chosen by triggering the *selected* event of the list. From the song list, the *selected* event fires the *Change song* action, which starts playing the chosen song; it also opens the *Playing* page, if this was closed, or refreshes the content of the *PlayerPlaying* component, which shows the currently playing song. The IFML DataFlow (dashed arrow) from *Songs* to the *PlayerPlaying* component represents the passage of the ID of the chosen song as a parameter from the source to the target of the link. The *Stop* event associated with the *PlayerPlaying* component fires the *Stop Song* action, which interrupts playing and opens the *Stopped* page. The *Stopped* page contains the *PlayerStopped* component, which displays the current song in the stopped status. Another IFML DataFlow represents the passage of the ID of the previously chosen song from the *Songs* component to the *PlayerStopped* component. The *Play* event, associated with the *PlayerStopped* component, fires the *Start Song* action, which restarts playing and opens the *Playing* page.

Fig. 12 shows the difference between the default UI generated by IFMLEdit.org and the final result.

Table 1 summarizes the main statistics about the implementation code of the applications, for the IFMLEdit.org and for the WebRatio architecture. The Quiz Game and Media Player are client-server applications, consisting of a front-end that interacts with back-end REST services over HTTP. The front-end implementation is different in IFMLEdit.org and WebRatio, due to the different presentation frameworks used by the two code generators. Also the realization of the business logic differs: IFMLEdit.org produces pure JavaScript (JS) applications whereas WebRatio uses Java.


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

Authors

Name
Daft Punk
The Beatles
Antonio Vivaldi

Songs

Author	Title
The Beatles	Let it be
The Beatles	Tomorrow never knows
The Beatles	Here come the sun

Now Playing

Title
Here come the sun

Stop

(a) Default (b) Custom

Fig. 12. Media Player: from prototype to final product

	JS		HTML		CSS		JSON		Java		XML	
	files	lines	files	lines	files	lines	files	lines	files	lines	files	lines
Quiz Game IFMLEdit	72	3067	29	259	1	290	5	651	0	0	0	0
Media Player IFMLEdit	30	1249	11	145	0	0	3	297	0	0	0	0
Quiz Game WebRatio	71	70519	11	341	4	1069	0	0	0	0	1	116
Media Player WebRatio	317	9133	13	1636	47	30641	0	0	5	612	8	440

Table 1. Code statistics of the Quiz Game and Media Player applications in IFMLEdit.org and in WebRatio

6 EVALUATION

In this section we address the question whether the Virtual Developer work-flow, which requires that part of the conflicts between the handwritten and generated code be solved manually, demands a "reasonable" amount of time. To answer, we compare the proposed approach with the process most used in practice: template-based code generation. We use for such a comparison two distinct development environments, so that the evaluation is less dependent on a specific code generation technology. We also assess the effect of simple transformation design guidelines for collision prevention, by comparing the original code generator of IFMLEdit.org with a new version that implements such collision prevention rules.

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

The use-case applications have been developed using WebRatio and IFMLEdit.org, with both model and code co-evolution and the template-based forward engineering processes. The two processes share the phases of model editing and feature development: in the former phase, the developer creates the model of the application and generates the code; in the latter, she programs the features that cannot be automatically generated from the model. The two processes diverge in the way in which the manually developed features are incorporated in the MDD loop, and in the effort required for this task. The template-based process requires the *back-porting*[15] of the manually programmed features into templates of the code generator, so that such features are automatically reproduced in the subsequent code generation steps. The model and code co-evolution process does not require the creation of additional artifacts: manually developed features are added on top of the generated code and automatically transported from one version to the next; the developer must work only on the parts of the code that produce collisions. In other terms, the work to integrate manually developed features in the template-based forward engineering process is proportional to the code-level size of the whole feature, whereas in the model and code co-evolution process it is proportional to the size of the part of the code where the automatically generated and the manually programmed code interfere: if the manually programmed feature is perfectly isolated from the automatically generated code, the integration work is null.

The tables in this Section compare the template-based and the co-evolution processes under the following aspects:

- *Feature development work*: the programming work that the developer devotes to implementing the application features; this has a different flavor in the two processes: in the template-based process, features are first implemented in the current version of the system and then back-ported into a template, so to incorporate them in the code generator; in the co-evolution process, features are implemented by manually modifying the generated version of the system; no back-porting is needed, but conflicts may arise at the next code generation round.
- *Integration work*: the "extra" work spent to integrate the developed features into the code generation loop; this has a different interpretation in the two processes: in the template-based process, it is the *back-porting* work

[15] With the term back-porting we refer to the act of migrating the changes manually added to a version of the system into the code generator templates, so that the next code generation produces a version identical to that modified by hand.

needed for creating or extending the templates; in the co-evolution process, it is the *conflict resolution* work needed to resolve the collisions between handwritten and generated code at the next generation round.

- *Total effort*: it is the sum of the feature development and of the integration work.

The feature development, integration, and total work are quantified by the number of lines of code (*atomic updates*) and of program features (*updates*), which the developer has to write or revise.

The integration work is evaluated as follows:

- In template-based forward engineering, we evaluate the effort for back-porting code-level features into the code generator; we compute the number of macro-updates (e.g., the implementation of an IFML abstract operation or of the services for data access, the modification of the visual style of an interface element, etc.) required to obtain the templates, and the number of lines involved.
- In model and code co-evolution, we evaluate the effort required for collision resolution, quantified by the number of collision groups, i.e., the number of developer's macro-interventions needed to perform conflict resolution, and by the number of atomic updates involved in the resolution of the collision groups, i.e., the number of lines requiring manual revision.

IFMLEdit.org: Table 2 and 3 show the results of comparing the template-based and the model and code co-evolution processes. Both the distribution and the amount of work differ in the two processes. The distribution of effort varies because the template-based process requires that customizations are performed *during the implementation of a sprint*, to generate the fully functional code; conversely, in model and code co-evolution, updates are applied to the generated code and thus the collisions are resolved *at the next invocation of the code generator*. This explains why the Proof of concept sprints have zero (collision detection) effort. The **integration** work due to collision resolution is less than 20%

	Model & Code Co-evolution				Template-based fw engineering			
	Feature development		Integration: conflict resolution		Feature development		Integration: back-porting	
Sprint	Updates	Atomic Updates	Updates	Atomic Updates	Updates	Atomic Updates	Updates	Atomic Updates
Proof of Concept	27	1154	0	0	28	1110	27	1102
Styling and Expl.	9	111	4	77	6	50	6	66
2^{nd} Mechanics	29	492	3	20	18	255	15	127
I18n	33	237	5	139	27	181	24	179
Total	98	1994	12	236	79	1596	72	1474

	Total effort (development + integration)		Total effort (development + integration)	
	Updates	Atomic updates	Updates	Atomic updates
	110	2230	151	3070

Table 2. Quiz Game: development statistics with IFMLEdit.org

	Model & Code Co-evolution				Template-based fw engineering			
	Feature development		Integration: conflict resolution		Feature development		Integration: back-porting	
Sprint	Updates	Atomic Updates	Updates	Atomic Updates	Updates	Atomic Updates	Updates	Atomic Updates
Proof of Concept	16	144	0	0	12	100	15	148
Styling	5	32	1	3	5	42	5	32
Songs Filtering	4	47	1	4	3	17	4	16
Songs Cover	6	34	2	10	3	14	2	9
Total	31	257	4	17	23	173	26	205

	Total effort (development + integration)		Total effort (development + integration)	
	Updates	Atomic updates	Updates	Atomic updates
	35	274	49	378

Table 3. Media Player: development statistics with IFMLEdit.org

of that required by back-porting of features into templates: the difference in the number of updates is 83% for the Quiz Game and 84% for the Media Player; the difference in the number of lines affected is 84% for the Quiz Game and 91% for the Media Player. To put such difference in context, Table 2 and 3 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: the number of updates required was reduced by 27% in the Quiz Game and by 28% in the Media Player, while the number of lines of code was reduced by 27% both in the Quiz Game and in the Media Player. The difference in the total 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 GUI templates, where achieving the desired visual effect is easier in the actual implementation code than in the “meta-level” code of the template.

WebRatio: Table 4 and 5 contrast the amount of work required by back-porting and collision resolution. The **integration** work due to collision resolution is less than 20% of that required by back-porting of features into templates: the difference in the number of updates is 100% for the Quiz Game and 83% for the Media Player; the difference in the number of lines affected is 100% for the Quiz Game and 96% for the Media Player. To put such difference in context, Table 4 and 5 show also the total application **development** work: the model and code co-evolution process required less than 65% of the effort required by the template-based process: the number of updates required was reduced by 45% in the Quiz Game and by 35% in the Media Player, while the number of lines was reduced by 38% in the Quiz Game and by 67% in the Media Player. Note that back-porting required the involvement of a tool expert, due to the proprietary nature of the template architecture; conversely, collision resolution was performed by a Java developer. 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.

	Model & Code Co-evolution				Template-based fw engineering			
	Feature development		Integration: conflict resolution		Feature development		Integration: back-porting	
Sprint	Updates	Atomic Updates	Updates	Atomic Updates	Updates	Atomic Updates	Updates	Atomic Updates
Proof of Concept	5	142	0	0	5	136	13	137
Styling and Expl.	4	29	0	0	4	24	4	22
2^{nd} Mechanics	10	42	0	0	4	11	4	8
I18n	5	15	0	0	5	16	5	16
Total	24	228	0	0	18	187	26	183
	Total Effort (development + integration)				Total Effort (development + integration)			
	Updates		Atomic Updates		Update		Atomic Updates	
	24		228		44		370	

Table 4. Quiz Game: development statistics with WebRatio

	Model & Code Co-evolution				Template-based fw engineering			
	Feature development		Integration: conflict resolution		Feature development		Integration: back-porting	
Sprint	Updates	Atomic Updates	Updates	Atomic Updates	Updates	Atomic Updates	Updates	Atomic Updates
Proof of Concept	2	21	0	0	2	21	4	25
Styling	6	47	0	0	5	48	6	153
Songs Filtering	2	4	2	7	0	0	0	0
Songs cover	1	2	0	0	1	2	2	4
Total	11	74	2	7	8	71	12	182
	Total Effort (development + integration)				Total Effort (development + integration)			
	Updates		Atomic Updates		Updates		Atomic Updates	
	13		81		20		253	

Table 5. Media Player: development statistics with WebRatio

6.2 Transformation design for collision prevention: guidelines and evaluation

The human effort for conflict resolution is proportional to the number of collision groups. Supposing that all the changes made by the human developers are needed, the only way to reduce conflict resolution effort is to decrease the number of collision groups. We provide a few minimalist transformation design guidelines to reduce the insurgence of collisions; these suggestions are conceived for the simplest, most popular, conflict detection approach: line-by-line source comparison. More sophisticated results can be obtained in two ways: 1) by making the transformation exploit more advanced code separation patterns, such as those presented in Section 2.1; 2) by employing more powerful code equivalence criteria, such as those discussed in Section 2.4.

Separation of concerns. The separation of areas that are the responsibility of the Virtual Developer and of the human developer can reduce collisions. Parts meant to be edited only by human developers may be generated, e.g., as code skeletons, to enable prototyping. Such mock-ups should follow a fixed template, so that the changes to their content appear only in manual revisions. They are introduced and deleted in generated revisions and edited only in manual revisions, thus the resolution of the conflict they produce can be automated. As aforementioned, the various techniques for separating the handwritten and generated code at the file and language construct level are discussed in the works presented in Section 2.1. Additionally, separation of concerns can be pushed to the line level. For example, in HTML-based GUIs as it is the case in our case studies, HTML white space invariability can be exploited to achieve line-level separation of concerns. HTML tags can be split from a single line to multiple lines containing different attributes. Attributes related to styling can be separated from attributes related to functional aspects or to data binding, so to better isolate changes to distinct aspects.

```
<span class="title" data-bind="text: item()['content']" ></span>
```

Reducing the size of changes. 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 stems from the non-deterministic iteration over model elements during code generation, due to the specifications of the transformation language or the exploitation of non semantically significant meta-data. Non-determinism was present in the initial version of the IFMLEdit.org tool, and is also found in mainstream commercial systems; for example, WebRatio allocates temporary identifiers non-deterministically in each code generation thread, some of which end up in the final code. Also, the internal representation of model element groups that carry no special meaning (e.g., the events and navigation flows associated with ViewComponents) is implemented with sets, which makes code generation non-deterministic. Also the Acceleo documentation acknowledges the problem and recommends best practices for generating code from collections.

Non-determinism 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']
};
```

	Original Implementation						Conflict Prevention					
	Manual		Generated		Collisions		Manual		Generated		Collisions	
Sprint	Update	Atomic Updates	Update	Atomic Updates	Update	Atomic Updates	Update	Atomic Updates	Update	Atomic Updates	Update	Atomic Updates
Proof of Concept	-	-	-	1442	-	-	-	-	-	1510	-	-
Styling and Expl.	27	1124	21	287	4	77	27	1054	16	274	1	37
2^{nd} Mechanics	30	1196	45	1168	3	20	30	1235	44	1209	1	6
I18n	47	1612	27	457	5	139	47	1709	25	404	1	6
				Total	12	236				Total	3	49

Table 6. Quiz Game: comparison of collisions without and with conflict prevention rules

	Original Implementation						Conflict Prevention					
	Manual		Generated		Collisions		Manual		Generated		Collisions	
Sprint	Update	Atomic Updates	Update	Atomic Updates	Update	Atomic Updates	Update	Atomic Updates	Update	Atomic Updates	Update	Atomic Updates
Proof of Concept	-	-	-	1202	-	-	-	-	-	1249	-	-
Styling	14	141	9	93	1	3	16	144	6	71	0	0
Songs Filtering	18	178	14	258	1	4	20	176	11	238	0	0
Songs Cover	22	217	12	303	2	10	24	223	5	268	2	14
				Total	5	17				Total	2	14

Table 7. Media Player: comparison of collisions without and with conflict prevention rules

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']
};
```

To understand the impact of the above-mentioned collision prevention guidelines, we revised the original code generator of IFMLEdit.org following such simple measures. The new version reduces non-determinism and enhances line-level separation of concerns. Table 6 and Table 7 show, for the original and for the enhanced version of the IFMLEdit.org code generator, the following data: the number of updates and of atomic updates of the human developer, the number of updates and of atomic updates of the Virtual Developer, the number of updates and of atomic updates required for collision resolution. The number of updates in $\Delta^{G}_{V,n+1}$ 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 conflict prevention guidelines in the code generator reduces the number of collision groups 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. To evaluate 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. Recalling from Table 2 and 3 that collision resolution is accountable for 10.9% of the total development effort for the Quiz Game and 11% of the Media Player, then collision prevention guidelines reduce the total development effort by 8% (75% of 10.9%) in the Quiz Game and 7% (60% of 11%) 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.

6.3 Discussion

The integration of manually programmed and automatically generated code is a long-standing issue, which affects the adoption of MDD. The model and code co-evolution approach addresses such a problem a posteriori, using the standard collision detection of VCSs to support the merge of the two types of code. It contrasts with template-based approaches, which are widely supported techniques of academic and industrial tools. In Section 4.4, we have shown a complete example of how the non-modeled features can be managed in the MDD process with the model and code co-evolution work-flow, by identifying and resolving collisions between the handwritten and the generated code; in this section we show how one of the customizations discussed in Section 4.4 can be achieved by templating. Next we show how the size of the collision can be reduced, by improving the code generator, so to enable automatic resolution. We conclude discussing the limitations of the model and code co-evolution approach and the further investigation needed for generalizing the reported results.

6.3.1 Template programming. Customizing the generated code can be achieved with a template-based approach. The following template, written in EJS for the IFMLEdit.org framework, applies to the *Song list* ViewComponent the same presentation obtained in the final revision ($R^M_{C,8}$) of Section 4.4.

```
<% var knownFields = ["title", "duration", "image"], // Fields at template development time
       templateFields = fields.slice(); // Make a copy of all model fields to sort them
   for (var field of knownFields) {
       var index = templateFields.indexOf(field);
       if (index === -1) continue; // If the field is in the list
       templateFields.splice(index, 1); // Remove it from the list
       templateFields.unshift(field); // Add it at the beginning of the list
   } -%>
<h3><%=name %></h3>
<table class="table table-hover table-condensed">
    <thead>
<% for (var i = 0; i < templateFields.length; i += 1) { /* For each field,  generate a column header */ -%>
            <th><%= templateFields[i]%></th>
<% } if (events.length) { /* If there are events, generate a  column */ -%>
            <th>actions</th>
<% } -%>
        </tr>
    </thead>
    <tbody data-bind="foreach: items">
        <tr data-bind="click: $parent.select">
<% for (var i = 0; i < templateFields.length; i += 1) { // For each field, generate the row cell
      switch (templateFields[i]) {
      case "image": -%>
            <td><img data-bind="attr: {src: $data['image']}" width="300px" class="img-thumbnail"></td>
<%        break;
      default: -%>
            <td data-bind="text: $data['<%= fields[i] %>']" style="white-space: pre-wrap;"></td>
<%    }
   } if (events.length) { /* If there are events, create a cell in the row */ -%>
            <td>
<%     for (var i = 0; i < events.length; i += 1) { /* For each event, generate a button */ -%>
                <button class="btn btn-primary" data-bind="click: $parent.trigger.bind($data, '<%= events[i].id %>'),
                      clickBubble: false"><%= events[i].name %></button>
<%     } -%>
            </td>
<% } -%>
        </tr>
    </tbody>
</table>
```

Note that the template contains more code than strictly necessary for producing the custom rendition of Section 4.4; it checks the model to determine if more fields have been added and positions them at the end of the table, preserving the required order of the fields for which the template is designed. It creates the HTML `img` element of the image field and renders the events associated with the component. The extra code is necessary to make code generation work robustly also if the *Song list* ViewComponent is altered; otherwise, the generator would produce code misaligned with respect to the model. The example also demonstrates that the dependency on aspects not reflected in the input model makes templates non-reusable. The order of elements in the interface, which is often relevant for usability, cannot be derived from the model, which is presentation agnostic on-purpose, and thus is hardwired into the template. An alternative, provided by most template-based MDD environments, is to provide custom annotations of the high-level models, or even aspect-specific sub-models, to capture the missing requirements. This makes template programming more complicated, because the template must extract information from multiple, possibly heterogeneous, inputs.

6.3.2 Limitations of the approach and of the case study. The suitability of model and code co-evolution as a simpler, yet effective, alternative to template-based code generation has been quantitatively assessed on realistic, yet small-scale, applications. Deeper studies on large, real world MDD projects, observed also during maintenance, are necessary to better quantify the pros and cons of both techniques. The reuse of templates within large projects and across different projects should be investigated, as an advantage potentially compensating the effort of template programming. Another element to consider is the shift of variability factors out of the code generation rules, in the spirit of [García et al. 2014], which could simplify the construction of templates. For example, WebRatio offers runtime services and configuration files, wrapped as custom server-side tags, which hide aspects that would otherwise complicate the code of the template, such as the input-output dependencies of components and the ordering of model elements. As a preliminary investigation of template reuse, we have considered the case study applications described in the paper. The average number of times that a template is reused for generating different model elements is 1.7 for the Quiz Game and 2 for Media Player. Furthermore, the inspection of a real world application[16], which publishes smart meter consumption data and suggestions for energy saving to consumers, shows that the number of templates necessary to customize the MDD code generator is 52 and the average number of reuses of a template on different model elements is 2.6.

As a final remark on the limitations of the case study, we note that a more comprehensive study would consider not only code-based metrics, but also time-based ones. We have not measured time, but coherently to the MDD literature, we observed that template programming, which is a kind of meta-programming, required more time, because developers are less familiar with it and debugging cannot be done directly on the template, but requires a template coding, code generation, and testing loop.

7 CONCLUSIONS AND FUTURE WORK

This paper presented an approach for model and text co-evolution, which allows a flexible interplay between manual programming and code generation in MDD work-flows. The idea is to manage the history of code revisions and to apply the same version control techniques normally used in distributed development also to the code generator, considered as a Virtual Developer. The proposed work-flow permits developers to identify and compare the changes introduced by

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

the last round of code generation and of manual feature programming and to exploit popular VCS primitives to support the reconciliation of conflicts stemming from the concurrent work of humans and tools. The described approach is implemented in an open source MDD environment and has been used to build a web and mobile application[17] for an energy demand management project, in which multiple versions of an energy awareness game have been produced in rapid sprints within a user-centric development process. Future work will focus on the following directions: 1) the support of multiple Virtual Developers and of conflict resolution also at the model level; 2) the integration of other code repositories and VCSs; 3) further comparison between template-based forward engineering and model and code co-evolution in larger industrial projects; we will consider factors not taken into account in the paper, such as the reuse of templates in multiple projects and the learning curve of template programming, to compare the total cost of both approaches in real scenarios; 4) the investigation of the potential effectiveness of model and code co-evolution to reduce the barriers for the adoption of MDD in companies, thanks to the elimination of template programming and the reduction of the gap between modeling and coding.

ACKNOWLEDGMENTS

This work has been supported by the European Community HORIZON 2020 project enCOMPASS (Grant #723059).

REFERENCES

Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. 2009. A survey on model versioning approaches. *IJWIS* 5, 3 (2009), 271–304. https://doi.org/10.1108/17440080910983556

Anthony Anjorin, Marius Paul Lauder, Michael Schlereth, and Andy Schürr. 2010. Support for Bidirectional Model-to-Text Transformations. *ECEASST* 36 (2010). https://doi.org/10.14279/tuj.eceasst.36.443

Michal Antkiewicz and Krzysztof Czarnecki. 2006. Framework-Specific Modeling Languages with Round-Trip Engineering. In *MoDELS 2006, Genova, Italy, Oct. 1-6, 2006 (LNCS)*, O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio (Eds.), Vol. 4199. Springer, 692–706. https://doi.org/10.1007/11880240_48

Carlo Bernaschina. 2017. ALMOsT.js: An Agile Model to Model and Model to Text Transformation Framework. In *Web Engineering - 17th International Conference, ICWE 2017, Rome, Italy, June 5-8, 2017, Proceedings (Lecture Notes in Computer Science)*, Jordi Cabot, Roberto De Virgilio, and Riccardo Torlone (Eds.), Vol. 10360. Springer, 79–97. https://doi.org/10.1007/978-3-319-60131-1_5

Carlo Bernaschina, Sara Comai, and Piero Fraternali. 2017. IFMLEdit.org: Model Driven Rapid Prototyping of Mobile Apps. In *4th IEEE/ACM Int. Conf. on Mobile Software Engineering and Systems, Buenos Aires, Argentina, May 22-23, 2017.* IEEE, 207–208. https://doi.org/10.1109/MOBILESoft.2017.15

Klaus Birken. 2014. Building Code Generators for DSLs Using a Partial Evaluator for the Xtend Language. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I (Lecture Notes in Computer Science)*, Tiziana Margaria and Bernhard Steffen (Eds.), Vol. 8802. Springer, 407–424. https://doi.org/10.1007/978-3-662-45234-9_29

Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-Driven Software Engineering in Practice, Second Edition.* Morgan & Claypool Publishers. https://doi.org/10.2200/S00751ED2V01Y201701SWE004

Antonio Cicchetti, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2013. Managing the evolution of data-intensive Web applications by model-driven techniques. *Software and System Modeling* 12, 1 (2013), 53–83. https://doi.org/10.1007/s10270-011-0193-0

Federico Ciccozzi, Antonio Cicchetti, and Mikael Sjödin. 2013. Round-trip support for extra-functional property management in model-driven engineering of embedded systems. *Information & Software Technology* 55, 6 (2013), 1085–1100. https://doi.org/10.1016/j.infsof.2012.07.014

Peter de Lange, Petru Nicolaescu, Thomas Winkler, and Ralf Klamma. 2018. Enhancing MDWE with Collaborative Live Coding. In *Modellierung 2018, 21.-23. Februar 2018, Braunschweig, Germany (LNI)*, Ina Schaefer, Dimitris Karagiannis, Andreas Vogelsang, Daniel Méndez, and Christoph Seidl (Eds.), Vol. P-280. Gesellschaft für Informatik e.V., 199–214. https://dl.gi.de/20.500.12116/14939

Jokin Garcia, Oscar Diaz, and Maider Azanza. 2013. Model Transformation Co-evolution: A Semi-automatic Approach. *SLE*, 144–163. https://doi.org/10.1007/978-3-642-36089-3_9

Jokin García, Oscar Díaz, and Jordi Cabot. 2014. An Adapter-Based Approach to Co-evolve Generated SQL in Model-to-Text Transformations. In *Advanced Information Systems Engineering - 26th International Conference, CAiSE 2014, Thessaloniki, Greece, June 16-20, 2014. Proceedings (Lecture Notes in Computer Science)*, Matthias Jarke, John Mylopoulos, Christoph Quix, Colette Rolland, Yannis Manolopoulos, Haralambos Mouratidis, and Jennifer Horkoff (Eds.), Vol. 8484. Springer, 518–532. https://doi.org/10.1007/978-3-319-07881-6_35

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

Jeremy Gibbons and Perdita Stevens (Eds.). 2018. *Bidirectional Transformations - International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures*. Lecture Notes in Computer Science, Vol. 9715. Springer. https://doi.org/10.1007/978-3-319-79108-1

Thomas Goldschmidt and Axel Uhl. 2013. Retainment policies - A formal framework for change retainment for trace-based model transformations. *Information & Software Technology* 55, 6 (2013), 1064–1084. https://doi.org/10.1016/j.infsof.2012.07.013

Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Pérez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. 2015. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *MODELSWARD 2015, Angers, France, Feb. 2015.*, Slimane Hammoudi, Luís Ferreira Pires, Philippe Desfray, and Joaquim Filipe (Eds.). SciTePress, 74–85. https://doi.org/10.5220/0005239700740085

Sandra Greiner and Thomas Buchmann. 2016. Round-trip Engineering UML Class Models and Java Models: A Real-world Use Case for Bidirectional Transformations with QVT-R. *IJISMD* 7, 3 (2016), 72–92. https://doi.org/10.4018/IJISMD.2016070104

Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. 2017. Approaches to Co-Evolution of Metamodels and Models: A Survey. *IEEE Trans. Software Eng.* 43, 5 (2017), 396–414. https://doi.org/10.1109/TSE.2016.2610424

Soichiro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. 2016. Feature-based classification of bidirectional transformation approaches. *Software and System Modeling* 15, 3 (2016), 907–928. https://doi.org/10.1007/s10270-014-0450-0

Bernhard Hoisl and Stefan Sobernig. 2015. Towards Benchmarking Evolution Support in Model-to-Text Transformation Systems. In *Proc. of the 4th Workshop on the Analysis of Model Transformations co-located with the 18th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2015), Ottawa, CA, Sep. 28, 2015. (CEUR Workshop Proceedings)*, Jürgen Dingel, Sahar Kokaly, Levi Lucio, Rick Salay, and Hans Vangheluwe (Eds.), Vol. 1500. CEUR-WS.org, 16–25. http://ceur-ws.org/Vol-1500/paper3.pdf

Steven Kelly and Juha-Pekka Tolvanen. 2008. *Domain-Specific Modeling - Enabling Full Code Generation.* Wiley. http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470036664.html

Gábor Kövesdán, Márk Asztalos, and László Lengyel. 2014. Polymorphic Templates: A design pattern for implementing agile model-to-text transformations. In *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. (CEUR Workshop Proceedings)*, Davide Di Ruscio, Juan de Lara, and Alfonso Pierantonio (Eds.), Vol. 1239. CEUR-WS.org, 32–41. http://ceur-ws.org/Vol-1239/xm14_submission_1.pdf

Tom Mens. 2002. A State-of-the-Art Survey on Software Merging. *IEEE Trans. Software Eng.* 28, 5 (2002), 449–462. https://doi.org/10.1109/TSE.2002.1000449

Andreas Pleuss, Stefan Wollny, and Goetz Botterweck. 2013. Model-driven development and evolution of customized user interfaces. In *ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'13, London, United Kingdom - June 24 - 27, 2013*, Peter Forbrig, Prasun Dewan, Michael Harrison, and Kris Luyten (Eds.). ACM, 13–22. https://doi.org/10.1145/2480296.2480298

Marcos Antonio Possatto and Daniel Lucrédio. 2015. Automatically propagating changes from reference implementations to code generation templates. *Information & Software Technology* 67 (2015), 65–78. https://doi.org/10.1016/j.infsof.2015.06.009

Iván Santiago, Álvaro Jiménez, Juan Manuel Vara, Valeria de Castro, Verónica Andrea Bollati, and Esperanza Marcos. 2012. Model-Driven Engineering as a new landscape for traceability management: A systematic literature review. *Information & Software Technology* 54, 12 (2012), 1340–1356. https://doi.org/10.1016/j.infsof.2012.07.008

Thomas Stahl and Markus Völter. 2006. *Model-Driven Software Development: Technology, Engineering, Management.* Wiley, Chichester, UK.

Eugene Syriani, Lechanceux Luhunu, and Houari A. Sahraoui. 2018. Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures* 52 (2018), 43–62. https://doi.org/10.1016/j.cl.2017.11.003

Axel Uhl. 2008. Model-Driven Development in the Enterprise. *IEEE Software* 25, 1 (2008), 46–49. https://doi.org/10.1109/MS.2008.12

Juan Manuel Vara, Verónica Andrea Bollati, Álvaro Jiménez, and Esperanza Marcos. 2014. Dealing with Traceability in the MDDof Model Transformations. *IEEE Trans. Software Eng.* 40, 6 (2014), 555–583. https://doi.org/10.1109/TSE.2014.2316132

Markus Völter and Jorn Bettin. 2004. Patterns for Model-Driven Software-Development. In *Proceedings of the 9th European Conference on Pattern Languages of Programms (EuroPLoP '2004), Irsee, Germany, July 7-11, 2004.*, Klaus Marquardt and Dietmar Schütz (Eds.). UVK - Universitaetsverlag Konstanz, 525–560. http://hillside.net/europlop/HillsideEurope/Papers/EuroPLoP2004/2004_VoelterEtAl_PatternsForModel-Driven.pdf

Stefan Winkler and Jens von Pilgrim. 2010. A survey of traceability in requirements engineering and model-driven development. *Software and System Modeling* 9, 4 (2010), 529–565. https://doi.org/10.1007/s10270-009-0145-0

A AN ILLUSTRATIVE EXAMPLE

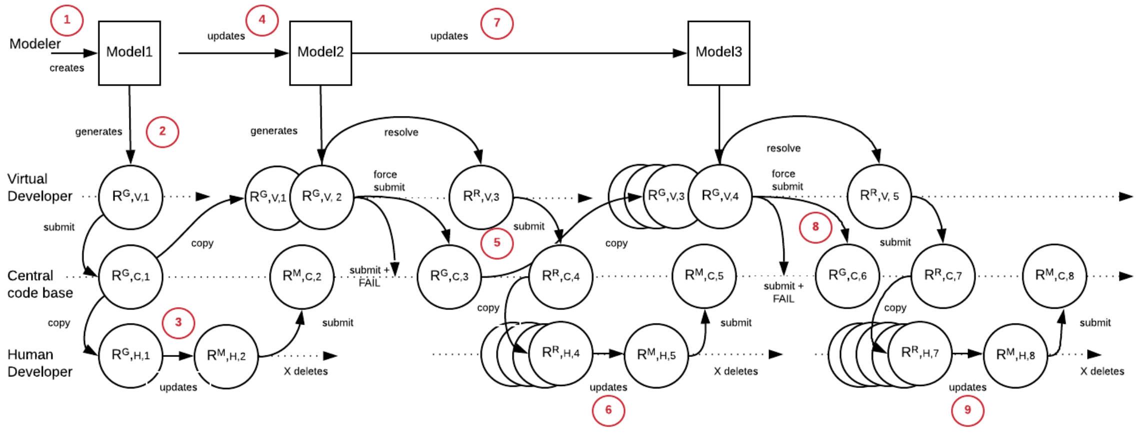


Fig. 13. Work-flow of the illustrative example

List of models

- Model 1: initial model, comprises a page displaying the list of songs, a form and an action for creating a new song. In this initial version, song are characterized by just a title and a duration.
- Model 2: the initial model is extended by adding a new attribute *image* to the song. This change requires the introduction of a new field in both the *New song* form and the *Song list* list, an input parameter in the *Create song* action and an extra binding in the flow targeting *Create song*.
- Model 3: the previous model is extended by adding the capability to delete a previously created song. A new *Delete Song* action is introduced having the id of the song that the user wants to delete as only input. A new *delete* event is introduced on the *Song list* list, with a flow connecting it to, and properly binding, the just introduced action.

List of revisions

- $R^G_{V,1} \rightarrow R^G_{C,1}$: the initial revision generated from Model 1; it is submitted to the central code-base initializing it. This initial revision will always be copied, among other, from the central code-base into the local code-bases initialized at the beginning of any change by both the human or the virtual developer.
- $R^M_{H,2} \rightarrow R^M_{C,2}$: after initializing a new local code-base, copying off the content of the central code-base up to the latest revision, the developer removes the id of the song from the list, implements the *Create song* action and the data access layer of the *Song* entity, generating a new revision of the local code-base. This revision is submitted to the central code-base without triggering any conflict.
- $R^G_{V,2} \rightarrow R^G_{C,3}$: after initializing a new local code-base, copying off the content of the central code-base up to the latest generated revision, the code generated from model 2 is introduced as a new revision of the local code-base; it adds the default markup to display the image field in the *Song list* view, the *image* input field in the *New Song* view and the comment to add the *image* parameter to the implementation of the *Create song* action. The initial submit of this revision fails due to the triggered conflict. After conflict resolution this revision will be force submitted to the central code-base.

- $R^{R}_{V,3} \rightarrow R^{R}_{C,4}$: the first conflict resolution revision; the manual changes to the *Create song* implementation, the *Song List* view and the data access layer, overwritten by the code generator, are reintroduced generating a new revision in the local code-base. Manual intervention is required on the *Create song* implementation due to the, advisable, collision. At the end of the resolution the previous generated revision is force submitted to the central code-base, while the current conflict resolution revision is submitted without triggering any conflict.
- $R^{M}_{H,5} \rightarrow R^{M}_{C,5}$: after initializing a new local code-base, copying off the content of the central code-base, the developer alters the order of the fields in the *Song list* view and customizes the appearance of the *image* field. This revision is submitted to the central code-base without triggering any conflict.
- $R^{G}_{V,4} \rightarrow R^{G}_{C,6}$: after initializing a new local code-base, copying off the content of the central code-base up to the latest generated revision, the code generated from model 3 is introduced as a new revision of the local code-base; it adds the implementation of the *delete* event in the *Song List* view and the default implementation of the *Delete song* action. The initial submit of this revision fails due to the triggered conflict. After conflict resolution this revision will be force submitted in the central code-base.
- $R^{R}_{V,5} \rightarrow R^{R}_{C,7}$: the second conflict resolution revision; the manual changes to the *Create song* implementation, the *Song List* view and the data access layer, overwritten by the code generator, are reintroduced generating a new revision in the local code-base. Manual intervention is not required because no collision is present between the changes introduced by the virtual developer and all the previously introduced manual changes. At the end of the automatic resolution the previous generated revision is force submitted to the central code-base, while the current conflict resolution revision is submitted without triggering any conflict.
- $R^{M}_{H,8} \rightarrow R^{M}_{C,8}$: after initializing a new local code-base, copying off the content of the central code-base, the developer alters the implementation of the *Delete Song* action and the implementation of the data access layer to address the new requirement. This revision is submitted to the central code-base without triggering any conflict.

List of deltas

- $\Delta^{M}_{C,2}$ = ($R^{M}_{C,2}$ - $R^{G}_{C,1}$): contains the lines modified by the first manual revision. These are the changes from the human developer that are considered during the conflict resolution which generates $R^{R}_{V,3}$.
- $\Delta^{G}_{V,2}$ = ($R^{G}_{V,2}$ - $R^{G}_{V,1}$) = ($R^{G}_{C,3}$ - $R^{G}_{C,1}$): adds the markup for the default display of the image attribute in the song list and the *image* parameter to the action. These are the changes from the virtual developer that are considered during the conflict resolution which generates $R^{R}_{V,3}$.
- $\Delta^{R}_{V,3}$ =($R^{R}_{V,3}$ - $R^{G}_{V,2}$) = $\Delta^{R}_{C,4}$ = ($R^{R}_{C,4}$ - $R^{G}_{C,3}$): restores the manual implementation of the *Create song* action, the removal of the id field in the *Song List* view and the changes to the data access layer. Most of the changes in $\Delta^{M}_{C,2}$, the ones not in conflict, have an equivalent counterpart in this delta. The delta also contains some new manual changes required to solve the, advisable, collision on the implementation of the *Create song* action.
- $\Delta^{M}_{C,5}$ = ($R^{M}_{C,5}$ - $R^{R}_{C,4}$): contains the lines modified by the second manual revision.
- $R^{M}_{C,5}$ - $R^{G}_{C,3}$: contains all the manual changes introduced so far. These are the changes from the human developer that are considered during the conflict resolution which generates $R^{R}_{V,5}$.
- $\Delta^{G}_{V,4}$ = ($R^{G}_{V,4}$ - $R^{G}_{V,3}$) = ($R^{G}_{C,6}$ - $R^{G}_{C,3}$): adds the markup for the *delete* event in the *Song List* view and the default implementation of the *Delete Song* action. These are the changes from the virtual developer that are considered during the conflict resolution which generates $R^{R}_{V,5}$.

- $\Delta^R_{V,5} = (R^R_{V,5} - R^G_{V,4}) = \Delta^R_{C,7} = (R^R_{C,7} - R^G_{C,6})$: restores the manual implementation of the *Create song* action, the alteration of the fields in the *Song List* view and the changes to the data access layer. Each change in $R^M_{C,5} - R^G_{C,3}$ has an equivalent counterpart in this delta.
- $\Delta^M_{C,8} = (R^M_{C,8} - R^R_{C,7})$: contains the lines modified by the third manual revision.
- $R^M_{C,8} - R^G_{C,6}$: contains all the manual changes introduced so far.