Fostering Software Architect and Programmer Collaboration

Anonymous Authors
Institution
Address
City, State, Country
Email

ABSTRACT

Model-Driven Engineering (MDE) is a development paradigm that brings the benefits of automation to a software development cycle. The MDE community tries to promote MDE adoption by pushing models written in diagram-based languages, supported by extensive tooling. While there are no more doubts that MDE fosters the design of complex software, its level of acceptance by software developers is still low. On one hand, rather than use diagram-based languages, most programmers prefer to work with their favorite textual programming language (e.g. Java and C++) and integrated development environment. On the other hand, software architects are among the early adopters and promoters of diagram-based languages. Such languages are indeed considered by the latter to be much more suitable for architecture description than textual languages. Synchronizing manually written artifacts in both forms of language is no simple feat and often very time-consuming without the correct automated methods and tools.

To solve this issue, this paper proposes a model-code synchronization methodological pattern and its underlying tooling. The solution tackles a classic problem of round-trip engineering: synchronization between concurrently evolved artifacts. On one hand we have the architecture model maintained by the software architects, and on the other hand, we have code manually written by programmers. Applying our approach for the development of a real runtime system, we show that both parties involved, software architects and programmers, can efficiently collaborate while continuing to work in their favorite development environment.

CCS Concepts

•**Software and its engineering** → *Software development methods; Collaboration in software development;*

ASE '16 Singapore

DOI: 10.475/123_4

Keywords

Round-trip engineering, Model-driven engineering, Model code co-evolution, UML, C++, Eclipse, Papyrus

1. INTRODUCTION

Model-Driven Engineering (MDE) [60] is a development paradigm that aims to increase productivity, in particular by promoting automation, a capability that is as much innate as required by MDE. Until now, a large part of the MDE community has clearly distinguished developing with high-level artifacts written in a diagram-based language - commonly called modeling language in the MODELS [48] community - and developing with low-level artifacts written in a textual language - commonly referred as code in a programming language. It is argued that the adoption of model-centric approaches would benefit companies from the many advantages [61] of MDE for complex system development. The community pushes for model-driven development, for example by promoting diagram-based languages with sufficient tooling support [24] like code generation.

There remains some reticence to adopt, all along the software development process, a completely model-centric approach in the industry [32, 70]. The reticence is due in part to the perceived gap [8] between diagram-based languages and textual languages. On one hand, software developers like programmers shy away from diagram-based languages and prefer to use their favorite programming language (i.e. a textual language) in an Integrated Development Environment (IDE). On the other hand there are developers like software architects, who favor the use of models, and are eager to use diagram-based languages to describe the architecture of the system.

Even in teams who embrace a model-driven approach, the gap between model and code is not entirely bridged. The survey in [33] questions stakeholders in companies who use MDE approaches. It reports that 70% of the respondents mainly make editions to their model but there is still manually-written code to be integrated. Furthermore, 35% of the respondents answered that they spend a lot of time synchronizing model and code.

The see-saw situation between diagram-based languages and textual languages, and between model and code, hinder the adoption of MDE in the industry. To solve this issue, we believe the clear distinction between model and code must be blurred. As such, we believe collaboration [44] solutions must be offered to different types of developers, with different development practices, without imposing a diagram-based language or a textual language, nor tooling.

Collaboration between developers producing different artifacts, in different languages, raises the issue of artifact synchronization, which is a typical concern of round-trip engineering. Indeed, round-trip engineering is the ability to automatically maintain the consistency of multiple evolving software artifacts, in a software development environment [4, 30, 62]. Round-trip engineering is related to traditional software engineering disciplines such as forward engineering [13] and reverse engineering [13]. Round-trip engineering adds synchronization of existing artifacts that evolve concurrently by incrementally updating each artifact to reflect editions made to the other artifact.

In this paper, we propose to use model-code synchronization as a mean to bridge the gap between models edited by software architects, and code written by programmers. Our proposition is based on the following contributions, exposed later in this paper:

1. A generic model-code synchronization methodological pattern to solve the problem of collaboration between different types of developers, when model and code coevolve. This contribution is based on several propositions:
   - A generic IDE with functionalities necessary for model-code synchronization. The functionalities are not dependent of a particular approach or technology.
   - Processes to use the IDE to synchronize model and code in several scenarios based on development practices defined in specific processes.
2. An Eclipse-based implementation of the approach for synchronization between UML models and C++ code.
3. Experience on applying the solution for the development of a real example.

Contrary to traditional solutions, we generalize our work by considering the case where the model is not only used for architectural design, but also for full implementation [12], as it is often promoted by the MDE community. The tooling of our solution focuses on the Unified Modeling Language (UML) [58], because it is the most popular software architecture description languages [46]. We wish to synchronize models in a diagram-based language like UML with code in a popular textual programming language like C++, used by 4.4 million [38] developers worldwide.

The work presented in this paper is motivated by the Papyrus-RT [11] runtime system which is developed in collaboration by companies CEA, Ericsson, and Zeligsoft. These companies are contributors to the MDE eco-system. The architecture of the system follows an object-oriented paradigm, and it was initially implemented as a C++ project with about 15000 lines of code. This joint effort raised the issue of collaboration between developers with either a MDE background or a more traditional background.

The rest of the paper is organized as follows. Section 2 defines the actors (roles) we consider in our work and the use-cases of the generic IDE for our model-code synchronization approach. In Section 3 we propose processes for synchronizing model and code, in several scenarios. A possible implementation of our solution, based on Eclipse technologies, is proposed in Section 4. Our propositions are evaluated in Section 5 through simulations and the case-study of Papyrus-RT runtime. Section 6 relates our work to existing research and industrial approaches. Finally, in Section 7 we conclude with some perspectives.

2. ACTORS AND USE-CASES

In this section we define the actors who will use our model-code synchronization approach to collaborate during development. Then we define the main use-cases expected from a generic IDE used by these actors. Some basic concepts related to the actors and use-cases are also defined in this section.

2.1 Actors and artifacts

In this paper we propose a model-code synchronization methodological pattern for collaboration between software architects and programmers. To generalize our approach, we will generalize the architect and programmer as actors with stark opposite development practices. We consider actors that represent two opposite types of developers in order to generalize our proposition to cases where model and code can both be used for the full implementation of a system, rather than just architectural design for the former, and algorithmic implementation for the latter. Before defining these actors, let us first introduce the concept of development artifact and baseline artifact.

Definition 1 (Development artifact). *A development artifact is a artifact, as defined in [52], that can be used for the full implementation of the system.*

For example a system can be entirely implemented as code. The code is then a development artifact. A model may also be a development artifact. It is then not only documentation of specification. For example a model can be used for implementation either by compiling it directly [12], or by generating code from the model, and compiling the code without the need to edit or complete the code. In our work, we assume that model and code are both development artifacts. A development artifact may be the baseline artifact, defined as follows:

Definition 2 (Baseline artifact). *A baseline artifact is one which may be edited manually. All other artifacts are produced from the baseline artifact through some process, and only through the process. Manual edition of artifacts other than the baseline artifact is forbidden.*

Actors called model-driven developer and code-driven developer are considered. The main difference between these two actors is what they consider as the baseline artifact.

Definition 3 (Model-driven developer). *A model-driven developer is an actor of the software development process for whom the model is the baseline artifact.*

Otherwise said, for the model-driven developer only the model should be edited manually. The code must always be produced from the model, usually automatically through some process that guarantees that the code is conform to the model. The software architect is a kind of the model-driven developer that edits the model to define/edit the architecture of the system. (S)he considers that the reference for the architecture should be the model.

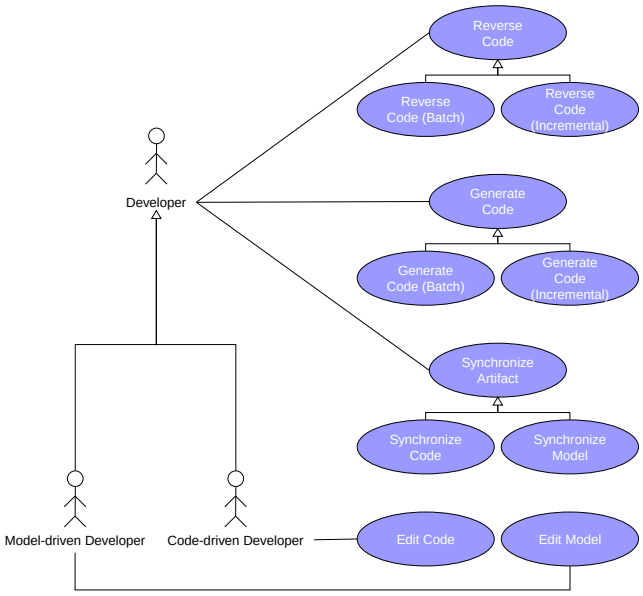

Figure 1: Use-case diagram: Notations of UML are used

Definition 4 (Code-driven developer). *A code-driven developer is an actor of the software development process for whom the code is the baseline artifact.*

The programmer is a specialization of the code-driven developer. Indeed, (s)he may for example edit method bodies. The code is then the main reference for the implementation of methods.

The next section presents the use-cases associated with these actors.

2.2 Use-cases

In this section we propose a generic IDE with use-cases that represent functionalities required by our model-code synchronization approach. Figure 1 shows a UML use-case diagram of the IDE and associations to the actors.

There are some use-cases for manual edition of artifacts. The `Edit Artifact` use-case means the IDE must have some tool to let the developer manually edit an artifact. The `Edit Model` and `Edit Code` use-cases are specializations of the `Edit Artifact` use-case where model and code are respectively the artifact in question.

There are also some use-case related to the synchronization of artifacts. The `Synchronize Artifact` use-case is the synchronization of two artifacts by comparing them, updating each artifact with editions from the other artifact, and reconciling conflicts, when appropriate, between the artifacts. The `Synchronize Model` and `Synchronize Code` use-cases are specializations where model and code are respectively the artifact to synchronize.

The `Generate Code` use-case is related to forward engineering [13]. It is the production of code in a programming language from a model. The developer can either use `Generate Code (Batch)` or `Generate Code (Incremental)`.

Definition 5 (Batch code generation [25]). *Batch code generation is a process of generating code from a model, from scratch. Any existing code is overwritten by the newly generated code.*

Incremental code generation is a specialization of incremental model transformation, which is defined in [25] as model transformation that does not generate the whole target models from scratch but only updates the target models by propagating editions in the source models.

Deduced from the definition of incremental model transformation, incremental code generation is defined in this paper as follows:

Definition 6 (Incremental code generation). *Incremental code generation is the process of taking as input a edited model, and an existing code, and then to update the code by propagating editions in the model to the code.*

Finally, the `Reverse Code` use-case is related to reverse engineering [13]. `Reverse Code` is the production of model, in a modeling language, from code, in a programming language. The developer can either use `Reverse Code (Batch)` or `Reverse Code (Incremental)`, defined in this paper as follows:

Definition 7 (Batch reverse engineering). *Batch reverse engineering is a process of producing a model from code, from scratch. Any existing model is overwritten by the newly produced model.*

Definition 8 (Incremental reverse engineering). *Incremental reverse engineering is the process of taking as input a edited code, and an existing model, and then to update the model by propagating editions in the code to the model.*

For readability purposes, in this paper sometimes we will designate batch and incremental as modes of code generation/reverse, e.g. we say that we generate code in batch mode from a model.

The use-cases are generic functionalities. They do not depend on any particular approach or tool. Therefore the software developers can choose the approach or tool that suits better his/her development preferences.

In the next section, the use-cases of the IDE will be integrated into some processes that cover model-code synchronization in several scenarios. The scenarios correspond to practices of both actors, i.e. model-driven developers and code-driven developers.

3. PROCESSES TO SYNCHRONIZE MODEL AND CODE

This section shows some processes to synchronize model and code within different scenarios. The scenarios are distinguished according to the type of developers, i.e. the actors, working on the system to develop.

We assume that the common starting point of each process, for each scenario, is the model. Therefore if there is some legacy code without a model, the legacy code must first be reversed into a new model in order to enter any of the processes. This can be done with the `Reverse Code (Batch)` use-case of the IDE proposed in Section 2.

Each of the following sections describes a scenario and the process associated with it for model and code synchronization.

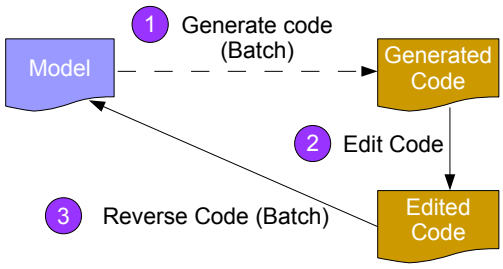

Figure 2: Synchronization process for scenario 1 where only code is edited: Dashed arrows are steps performed only once. Filled arrows are repeating steps.

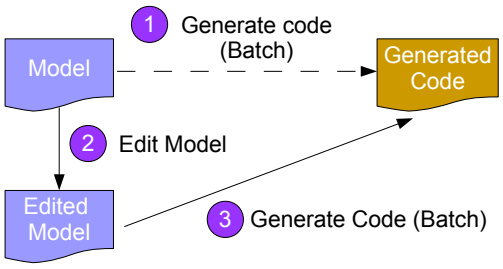

Figure 3: Synchronization process for scenario 2 where only model is edited: Dashed arrows are steps are performed only once. Filled arrows are repeating steps.

3.1 Scenario 1: editions only made in code

The assumption in scenario 1 is that the code is the only baseline artifact. (Otherwise said, only the code may be edited manually.) We assume that during development, the model needs to be synchronized sporadically with the code. This scenario usually suits better the code-driven developer.

Figure 2 shows the process to synchronize code and model in scenario 1. Note that each step is a use-case of the IDE proposed in Section 2. The general approach is to always overwrite the model with a new model produced from the edited code.

In Figure 2, as a reminder of the general assumption, the starting point of the process is the model. The steps of the process are described as follows:

- Scenario 1 synchronization process steps -

Step 1 Code is generated in batch mode from the model to begin implementation. The developers then work with this generated code.

Step 2 The code is edited.

Step 3 After the code has been edited, it is reversed in batch mode, i.e. the existing model is overwritten by a new model reversed from the code.

Obviously code can evolve several times through successive editions. After each edition, it can be reversed in batch mode to the model. Therefore steps **(2)** and **(3)** are repeating steps in the process.

Since the code is the baseline artifact, we only need to overwrite the model with the code, by batch reverse, for both artifacts to be synchronized. Using incremental code reverse has the same effect as batch code reverse.

3.2 Scenario 2: editions only made in model

Scenario 2 is the opposite of scenario 1: the assumption here is that the model is the only baseline artifact. Otherwise said, only the model may be edited manually. This scenario suits the model-driven developer.

Figure 3 shows the process to synchronize code and model in scenario 2. The general approach is to always overwrite the code with new code produced from the edited model. As a reminder, we assume that the model is a development artifact so it contains enough information for the full implementation of the system.

Steps of the process shown by Figure 3 are described as follows:

- Scenario 2 synchronization process steps -

Step 1 Code is generated in batch mode from the model to begin implementation.

Step 2 The model is edited.

Step 3 After the model has been edited, code is generated in batch mode, i.e. the existing code is overwritten by new code generated from the model.

Each time we edit the model, code is generated from it in batch mode. Therefore steps **(2)** and **(3)** are repeating steps in the process.

Since the model is the baseline artifact, we only need to overwrite the code with the model, by batch generation, for both artifacts to be synchronized. Using incremental code generation has the same effect as batch code generation.

3.3 Scenario 3: concurrent editions

Note that the process proposed for scenario 1 and scenario 2 are classic cases of forward and reverse engineering, rather than a full round-trip engineering case with the issues of synchronization. Indeed, code is produced from model in batch mode. When it is reversed, it is revered to the model in batch mode. There is no need for extra synchronization strategies since only code or model is the baseline artifact, i.e. only one is edited manually and there are no concurrent editions. Therefore batch code generation and reverse are sufficient for synchronization.

In scenario 3, there is no unique baseline artifact, therefore both the model and the code may be edited manually. Therefore they may evolve concurrently during development activities and synchronization issues may be raised. This scenario tackles the problem where model-driven and code-driven developers collaborate. Figure 4 shows the process to synchronize code and model in scenario 3.

Steps of the process shown by Figure 4 are described as follows:

- Scenario 3 synchronization process steps -

Step 1 Like in the other processes, code is generated in batch mode from the model.

Step 2.a and 2.b Afterwards both the model and code may be edited.

Step 3 After both artifacts have been edited concurrently, we need to synchronize these artifacts.

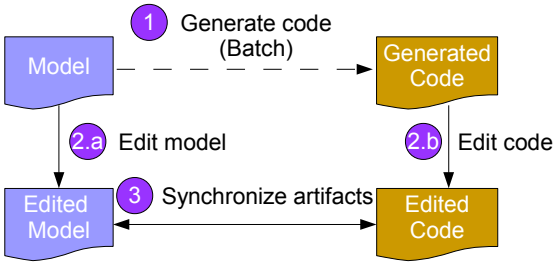

Figure 4: Synchronization process for scenario 3 where model and code are concurrently edited: Dashed arrows are steps are performed only once. Filled arrows are repeating steps.

Figure 5: Synchronization strategy 1 using code as synchronization artifact

In this paper we propose two synchronization strategies. The general approach behind our synchronization strategies is to represent the one artifact in the language of the other artifact. These artifacts can then be compared. Let us define a concept of a synchronization artifact within the scope of our work:

DEFINITION 9 (SYNCHRONIZATION ARTIFACT). *A synchronization artifact is an artifact used to synchronize model and code. It is an image of one of the artifacts, either model or code. In this paper, an image I of an artifact A is a copy of A obtained by transforming A to I. A and I are in different languages.*

For example a synchronization artifact can be code generated from the edited model in batch mode. In this case it is code that represents an image of the edited model.

Using the concept of synchronization artifact, two strategies are proposed in this paper: one where the synchronization artifact is code, and one where the synchronization artifact is model. We propose two strategies so the developer can choose to either use the `Synchronize Code` or `Synchronize Model` use-cases of the IDE. His choice may be due to development practices, or the availability of efficient tools.

Figure 5 shows the synchronization strategy based on using code as the synchronization artifact.

The steps of the process shown in Figure 5 are described as follows:

- Steps of synchronization strategy 1 -

Step 1 First we create a synchronization artifact from the edited model by generating code in batch mode. This synchronization artifact is code and it is an image of the edited model.

Step 2 The synchronization artifact is synchronized with the edited code. Since the synchronization artifact is code itself, this step is done with the use-case `Synchronize Code` of the IDE.

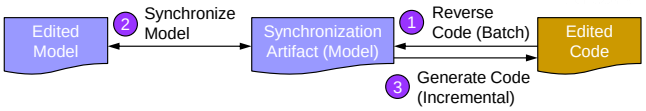

Figure 6: Synchronization strategy 2 using model as synchronization artifact

Step 3 Once synchronization artifact and edited code are synchronized, the former is reversed incrementally to update the edited model.

Figure 6 shows the synchronization strategy based on using model as the synchronization artifact. This strategy is the opposite of the strategy presented in Figure 5. Its steps are described as follows:

- Steps of synchronization strategy 2 -

Step 1 The synchronization artifact is obtained by reversing the edited code in batch mode.

Step 2 Afterwards the synchronization artifact is synchronized with the edited model.

Step 3 Finally, we generate code incrementally from the synchronization artifact to update the edited code.

We propose two strategies based on the preferences of the developer. (S)he may also use both strategies, successively, as a hybrid strategy. This may be useful when the developer wants to synchronize parts of the system with one strategy, and other parts with the other strategy. For example, (s)he may choose to synchronize method bodies using strategy 1 based on a synchronization artifact that is code. Then strategy 2, based on a synchronization artifact that is model, is used to to synchronize architectural elements.

In the next section we propose an implementation of the IDE and synchronization processes, thus offering tooling for our approach.

4. IMPLEMENTATION TO SYNCHRONIZE UML MODEL AND C++ CODE

Through tooling, the model-code synchronization approach can be automated. We propose an Eclipse-based implementation of the IDE proposed in Section 2. The implementation is used in the synchronization processes proposed in Section 3. Our implementation targets synchronization of Object Management Group (OMG) standard UML 2 [53] and C++11 code.

The use-cases of the IDE are implemented with some Eclipse technologies. **Eclipse CDT** is an IDE for C++ development. It is used to `Edit Code`.

Eclipse **Papyrus** [24] is used to `Edit Model`. Papyrus is a UML modeler that uses the Eclipse Modeling Framework (EMF) [65] implementation of the OMG-standard UML 2. Papyrus supports UML profiles for domain-specific modeling, and we may use the UML profile of Papyrus dedicated to C++ to facilitate and accelerate modeling of some C++ features.

We developed plugins for Papyrus to `Generate Code` from UML to C++ and to `Reverse Code`. The batch modes of

these use-cases does not need additional technologies to implement. For use-cases `Generate Code (Incremental)` and `Reverse Code (Incremental)`, we choose to listen to modification events in the model and code respectively. Listening to modification events is one possible approach in incremental model transformation [43]. The **Viatra** [5] API is used to listen to such events in the model. The Eclipse CDT API is used to listen to modification events in the code. These list of events are used to either generate code or reverse code incrementally.

EMF Compare [9] is used to `Synchronize Model` implemented with EMF. We adapted EMF Compare for for Papyrus in order to synchronize UML models for our specific work. Eclipse CDT is used to `Synchronize Code` with its built-in C++ features.

The next section exposes some experiments done using the implementation of our synchronization solution.

5. EXPERIMENTS

This section reports our experiments with the proposed model-code synchronization approach and its implementation based on Eclipse technologies. The contributions are applied for UML and C++. Two experiments have been conducted in order to assess the proposed methodology and its applicability to the development of a real system. In the following sections, we first expose some simulation results, operated to test the proposed approach. Then we report on a real case-study used to validate some scalability and usability points.

5.1 Simulations

Simulations have been conducted to test that our proposition respects the round-trip engineering laws [20], namely *right-invertibility* and *left-invertibility*. These laws are stated as follows:

Law 1: Right-invertibility means that not editing the code (respectively model) shall be reflected as not editing the model (respectively code). The model used for generating code and the model received by immediately reversing the generated code, without any editions to the code, must be the same.

Law 2: Left-invertibility means that all editions on the code are captured and correctly propagated to the model so that the edited code can be generated again by applying code generation on the updated model.

In the following sections first the model generator used for the simulations is presented. Afterwards simulations for both aforementioned laws are exposed. Finally, we simulate the process for scenario 3 (Section 3.3) where both model and code are edited concurrently.

5.1.1 Randomly generated UML models

In the simulations, UML models are randomly generated with a model generator that can be configured. The generated models have C++ features represented in UML. The generator can be configured to generate an average number of each C++ feature represented as a UML element.

Three packages are generated for each model. Each package contains 60 classes, and in average 10 enumerations, 10 structures, and 10 function pointers. Class members include methods with parameters, and attributes.

Types of parameters and attributes are chosen randomly. Methods have randomly generated bodies that use other classes. Attributes have randomly generated default values conform to their types.

Relationships between classes are also generated: associations, inheritances and dependencies. When dependencies are generated, the generator enforces that the source class of the dependency has a method that uses the target class of the dependency.

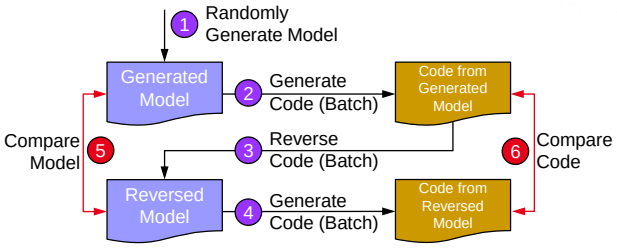

Figure 7: Simulation 1 for Law 1 right-invertibility

5.1.2 Simulation for Law 1 right-invertibility

In the first simulation, Law 1 is evaluated. The procedure for the simulation is shown in Figure 7.

The general idea behind the simulation procedure is to do a full round-trip of some UML model randomly generated in step **(1)**. The round-trip of the model is done through steps **(2)** to **(3)**. The randomly generated model is compared with the reversed model after the round-trip. This is done in step **(5)**.

A full round-trip is also done for the C++ code that is generated from the original randomly generated model in step **(2)**. The round-trip is done through steps **(3)** to **(4)**. The code generated from the original randomly generated model is compared with code generated from the reversed model in step **(6)**.

Code generation and reverse generation are done in batch mode since no editions are made to the models or code in this simulation. Comparison of models is done by first comparing the number of each type of elements. Then if the numbers of each type of elements is the same, EMF Compare is used. Comparison of code is done by using the built-in code comparison tool of Eclipse CDT.

Table 1 shows the number of each type of elements in the randomly generated model, and the comparison results, for 3 of the 200 models created by the generator. After launching the simulation for the 200 models, no differences were found during model comparison and code comparison. This result tests that our model-code synchronization approach satisfies Law 1 of round-trip engineering.

5.1.3 Simulation for Law 2 left-invertibility

In the second simulation, Law 2 is evaluated. The procedure for the simulation is shown in Figure 8.

The general idea behind the simulation procedure is to do a full round-trip of a randomly generated UML model but with editions introduced to the generated C++ code. As shown in Figure 8, the procedure to test our framework consist of the following steps:

Step 1 A UML model is randomly generated and becomes the **generated model**.

Step 2 Through batch code generation, we obtain **code**

Table 1: Three of 200 generated models for simulation 1: Abbreviations are classes (C), methods (M), method parameters (P), attributes (A), associations (AS), inheritances (G), method body (B), dependencies (D), enumerations (E), structures (S), default values (DV), function pointers (F)

Model ID	C	M	P	A	AS	G	B	D	E	S	DV	F	Differences in models?	Differences in code?
1	180	1419	2683	1835	1053	179	1138	507	13	7	150	11	No	No
2	180	1437	2718	1874	1074	179	1157	512	10	9	167	9	No	No
..	..	..	..	..	..	..	..	..	..	..	..	..	No	No
200	180	1413	2629	1857	1018	179	1127	517	13	9	150	13	No	No

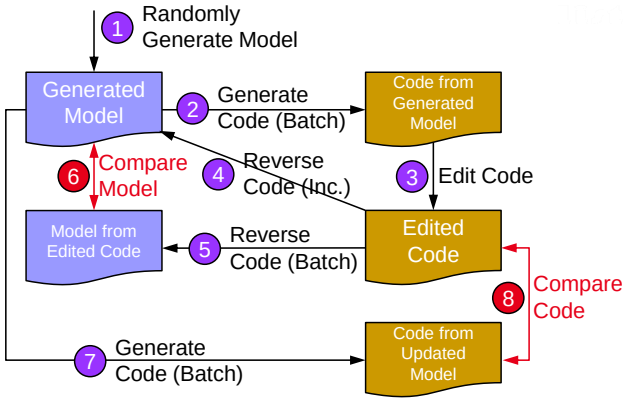

Figure 8: Simulation 2 for Law 2 left-invertibility

from generated model.

Step 3 The code produced from generated model is then edited. We thus obtain **edited code**. The different kinds of editions will be described when we discuss the results of this simulation.

Step 4 The **edited code** is reversed incrementally to the **generated model**. The **generated model** then becomes an **updated model**.

Step 5 The **edited code** is also reversed in batch mode to a **model from edited code**. This model is an image of the edited code.

Step 6 The **updated model** (previously **generated model**) is compared to the **model from edited code** (image of the edited code).

Step 7 Afterwards, we also generate in batch mode **code from the updated model**.

Step 8 The **code from the updated model** is compared to the **edited code**.

The simulation is run for 200 randomly generated models. During the simulations, the code generated from each model undergoes 7 kind[illegible]edition independently (the kinds of edition are described [illegible]Table 2). The editions trigger three kinds of Eclipse CDT modification events: `ADDED`, `REMOVED`, and `CHANGED`. Events `ADDED` and `REMOVED` mean addition and respectively deletion of classes, attributes or methods to the code. Event `CHANGED` is the update of elements in code including renaming attributes, classes, methods, changing the type of attributes, parameters, or changing the behavior of methods.

Table 2: Change description: Abbreviations are `ADDED` (A), `CHANGED` (C), `REMOVED` (R)

Edition kind	A	C	R	Model diff?	Code diff?
Renaming attributes of all classes	0	1903	0	No	No
Renaming methods of all classes	0	1197	0	No	No
Deleting attributes	0	0	46	No	No
Adding attributes	25	0	0	No	No
Adding methods	10	0	0	No	No
Changing method body	0	30	0	No	No
Mixing all editions	39	34	30	No	No

Table 2 shows the average number of `ADDED`, `REMOVED`, and `CHANGED` events that are triggered by each of the 6 kinds of edition. The last two columns show the results of model comparison and code comparison for all 200 simulations after each kind of edition is made independently.

5.1.4 *Simulation for concurrent edition of model and code*

A third [illegible]ulation aims at emulating the process described in Figure [illegible]ection 3.3, where model and code are concurrently edited. The process is simulated for 200 randomly generated UML models and their generated C++ code.

To simplify the simulation, the simulator only introduces attribute renaming editions model-side. Code-side, the simulator only introduces method body editions. To synchronize edited model and edited code, the strategy using code as synchronization artifact is cho[illegible] As a reminder, this strategy is described in Figure 5.

The simulator propagates all method body editions from the edited code to the synchronization artifact. The simulator then propagates all attribute renaming editions from the synchronization artifact to the edited code. Afterwards, the synchronization artifact is reversed incrementally to the edited model. At this point the model and the code are considered synchronized.

To assess that both synchronized model and code are im-

ages of one another, the synchronized code is reversed in batch mode to a new model. The new model is compared with the synchronized model. No differences were found during comparison. We also generate new code in batch mode from the synchronized model. The new code is compared to the synchronized code. No differences were found during comparison.

The same simulation is repeated for the case where model is used as synchronization artifact. Again the synchronized code and model are images of one another. Therefore with this example of concurrent edition of model and code, our proposition is tested.

Now that our model-code synchronization approach has been tested through all three simulations, we apply the solution for the development of a real system. We report on this experiment in the next section.

5.2 Case-study

In Section 5.1 our proposition was assessed for UML and C++ synchronization, through simulations. In this section we apply the whole synchronization approach on a case-study, in order to evaluate:

- The usability of the proposed synchronization approach
- The scalability of the tooling to a real system

The case-study is related to the development of the runtime underlying Papyrus-RT [11]. This latter is a modeler that relies on UML extensions for the design of real-time systems. Papyrus-RT features code generation for UML models. The generated code executes on the specific runtime that provides a C++ support to the high-level concepts provided by the domain specific modeling language available in the Papyrus-RT tool.

At the beginning of the project, the resource available to develop the aforementioned runtime was an experienced C++ programmer. For obvious pragmatic reasons of project management, it was decided to implement the runtime in C++ without the benefits of models. However, later it was concluded that even if the result was good, it was a shame to not be able to benefit from MDE assets. Indeed, MDE approaches would improve both maintainability and evolvability of the runtime. Moreover, in the meantime, the project team has grown including new developers that are proponents of MDE paradigms and are willing to work with models.

Therefore this case study was an ideal case study to assess our propositions first in terms of its usability, but also in terms of its scalability.

The runtime was originally implemented as an open-source plain C++11 project. Most of its architecture is object-oriented. The runtime has 65 classes and 14945 lines of code. Other than containing typical entities found in object-oriented architectures, the runtime uses C/C++ features such as type definitions, templates, pointers, references, function pointers, and variadic functions to name a few. These features are supported by the the reverse and code generation tools in Papyrus, coupled with the C++ UML profile.

In order to use our model-code synchronization approach, the original Papyrus-RT runtime had first to be reversed to a UML model. This step was crucial because the the original runtime contained some elements that were not object-oriented. Some modifications and refactoring were required for the original runtime to be entirely object-oriented. It can then be entirely modeled in a language like UML. Table 3 shows the two main differences between the original runtime, and the object-oriented runtime.

Table 3: Differences in Papyrus-RT runtimes

Original	Object-oriented
Directives to control compilation of OS-dependent code	Abstract OS-independent classes inherited by OS-dependent classes with implementation
Variables, functions and type definitions out side of class	Refactored as entities inside a class. Attributes and functions are static with visibility defined according to scope of original variable and function

The reverse in batch mode of the object-oriented runtime takes about 12 seconds. All 65 classes were reversed, with all of their attributes and methods. Code generation in batch mode of the entire reversed UML model takes about 5 seconds and produces 22053 lines of code. The difference in the number of lines of code is due to automatically generated documentation comments. The generated runtime compiles and the updated existing unitary tests pass when applied on the runtime compiled from generated code.

Once the runtime has been reversed, our model-code synchronization processes could be used. We noticed that using our approach introduced MDE approaches into the development of the Papyrus-RT runtime. This brought several advantages to the development as several tasks were then automated:

- Automatic handling of relationships between model elements, i.e. association, dependency, inheritance
- Automatic generation of includes and forward declarations in code that avoids cyclic dependencies
- Graphical representation of architecture in automatically updated UML diagrams (feature of Papyrus)

In conclusion, in order to apply our synchronization approach and collaborate in the development of the runtime, we only faced the difficulty of actually initializing the synchronization processes. Indeed, this required the reverse engineering of the original runtime, with some non-object-oriented code, into an object-oriented UML model. The reverse was done successfully after some modifications and refactoring. Once the whole model-code synchronization process was in place, we were able to use it and develop with the UML model and its generated C++ code. The development of the runtime then benefited from automation through MDE.

In the following section we relate both our work, including the implementation, to existing approaches and tools presented in the literature.

6. RELATED WORK

Our work is motivated by the will to reduce the gap between model and code, between diagram-based languages and textual languages. We use synchronization as a mean

to achieve this goal. ~~In the following sections, we compare our work to works in the literature related to such topics.~~

6.1 Comparison of diagram-based and textual languages

Diagram-based languages is a subset of visual languages, also called graphical languages. Several works have opposed textual languages to visual languages.

Many works [7, 49, 50, 57, 59, 69] discuss the pros and cons of visual languages, compared to textual languages, in different contexts. Some works [8, 10] also discuss what limits the adoption of visual languages.

Contrary to these works, we do not strictly oppose textual languages to visual languages like diagram-based languages. Rather we wish to blur the frontier between such languages. The idea that visual languages can co-exist with textual languages is indeed already expressed in several other works.

In [14] the author argue that the gap between textual and sual languages is narrow. They propose a framework to represent code in a visual language to improve comprehension of the code.

In [17] the authors propose a generic framework to use both visual and textual languages at the same time. The visual artifact is translated to textual code. Much importance is given to the human factor, i.e. the framework can be customized based on the preferences of the developer.

In [42] the authors propose a visual and textual language for the 3D animation domain. It is up to the system to inform a user, in text form, how his or her visual operation is interpreted.

In domains such as requirements engineering, tools as IBM Rational Doors [34] and Visure Requirements [67] are conscious of the importance of supporting requirements written traditionally in plain text by developers. These tools allow the transformation of requirements written in a textual human language to use-cases and structured requirements.

Contrary to our proposition, these works are not applied to the specific case of models and code that can be used for full implementation of the system. Usually one of the artifact is only used for better comprehension. There is no need for synchronization between artifacts because there is only one-way transformation.

Our argument that, in order to be more efficient, developers should not be imposed a diagram-based or textual language, is related to of software comprehension. Works [45, 68] that emphasize the role of software comprehension, for efficient software maintenance and evolution, date back to as far as the late eighties/early nineties. In our work we consider software architects and programmers. The former foster diagram-based language to architecture description and comprehension. The latter prefer textual languages considered much more powerful than diagram-based languages for algorithmic implementation for example.

6.2 Artifact synchronization

Our work is also closely related to synchronization of different artifacts used for development. In the following paragraphs, our model-code synchronization approach is compared to works on model-code round-trip engineering works, viewpoint synchronization, and model synchronization.

Round-trip engineering of model and code.

Several commercial and open-source tools [1–3, 23, 35, 51, 56, 63, 64] have support for round-trip engineering between UML model and code. Systematic reviews of some of these tools are available in [15, 28]. Support for Java round-trip is prominent in most tools. Other languages such as C++ are only available in a few [2, 23, 63]. Our methodological pattern does not focus on a particular programming language or a particular modeling language. Furthermore, the implementation of our approach is dedicated to UML and C++, which is less supported by these tools than Java. Usually these tools only support architectural elements model-side. The model cannot be used for full implementation and dependencies derived from method bodies are not considered during the round-trip. In our work, we assume that the model can be used for full implementation. Furthermore, our implementation analyzes C++ method bodies not only to reverse them to UML, but also to derive dependencies in the UML model. Some tools [2, 63, 64] only allow one of the artifacts, model or code, to be edited at a certain time. There is then no problem of synchronizing model and code that evolve through concurrent editons. Finally, some tools [23] do not support a real incremental reverse or code generation, by omitting modifications events such as a change, instead of a deletion then addition.

Some round-trip engineering techniques restrict the development artifact to avoid synchronization problems. Partial round-trip engineering and protected regions are introduced in [22, 39] Such techniques aim to preserve code editions which cannot be propagated to models. This approach separates the code regions which are generated from models from regions which are allowed to be edited by developers. This form of round-trip engineering is unidirectional and does not support iterative development [36]. Fujava [41] offers a round-trip engineering environment for UML and Java. Round-trip engineering is limited to elements following a naming convention. In our work there are no restrictions to editions on model and code, and we do not impose naming conventions.

Viewpoint synchronization.

Model and code can be both seen as different viewpoints [18] of the system to develop. Viewpoints partition the model of the system into several representations. Synchronization between viewpoints is crucial to maintain their consistency.

In [16] the authors improve the modeling of relationships and constraints between elements in different viewpoints in order to better guarantee the consistency of viewpoints. In [27] the authors argue that inconsistencies will exist in systems developed with different actors, using different viewpoints. They suggest that tools must exist to tolerate inconsistencies. A distributed graph transformation is proposed to the problem of formalizing the integration of multiple viewpoints in software development. Their work focuses on requirements engineering. Contrary to these works, our solution targets specifically model and code. The latter is not usually considered in viewpoint synchronization due to its low-levelness. Furthermore, our approach does not require explicit modeling of relationships between model and code elements.

Model synchronization.

Viewpoints synchronization is generalized by model synchronization for which there is an abundance of techniques

presented in the literature. Model synchronization aims to maintain consistency between a source model and a target model. Model synchronization works are usually categorized by their model transformation which can be total, injective, bi-directional, or partial non-injective [31].

Many model synchronization techniques require the explicit mapping of source model and target model. The authors in [55] propose an injective mapping of elements in the source model to the target model. The mapping can be used for synchronization. Techniques and technologies like Triple Graph Grammar (TGG) [26, 40] and QVT-Relation [54] allow synchronization between source and target elements who have non-injective mappings. The authors in [29] formalize TGG for synchronization of models that are concurrently edited. All of these techniques require a mapping model to connect source and target models with typed traceability links which need to persist in a model store [6]. This means that editing one model requires the presence of the other one. Our model-code synchronization approach does not require a mapping model and artifacts may be edited independently of the presence of the other artifact.

Other techniques [21, 66, 71] are based on bi-directional transformations which is the forward transformation of source model to target model, and the backward transformation of target model to source model. Bi-directional transformations provides a novel mechanism for synchronization [19]. Indeed, some works [19, 47] derive a backward transformation based on forward transformation. Such works do not offer any means to synchronize models that are concurrently edited.

Some works derive model synchronization from model transformation while allowing concurrent editions of both source and target models. In [72] the authors propose to automatically derive model synchronization of a source and a target model related by ATL [37] model transformation. The synchronization is based on differentiating source and target model states. Reflectable addition of an element in the target model is not well handled according to [72]. Our approach is generic and does not depend on a certain technology. Furthermore, in our implementation we propose to use modification events rather than state differences for incremental transformations, necessary for synchronization.

As a final note, we argue that our methodological pattern is generic. Therefore, many synchronization techniques found in the literature can be integrated into our approach as, for example, the `Synchronize Model` or `Reverse Code` use-cases of the proposed generic IDE. Finally, our solution targets collaboration between software architects and programmers who wish to use diagram-based languages or textual languages. Therefore we propose to use a synchronization artifact in the preferred language of the developer, rather than directly synchronizing artifacts in different languages.

7. CONCLUSION

A generic model-code synchronization methodological pattern was presented in this paper. The proposition targets collaboration between software architects and programmers, without imposing to either developers a diagram-based language or a textual language. An Eclipse-based implementation of the approach was presented for OMG-standard UML and C++11 synchronization.

Simulations assessed our synchronization approach with respect to both laws of round-trip engineering, and validated it. We also simulated scenarios where model and code are concurrently edited, then synchronized with our approach. The simulations let us test our proposition in such cases.

The approach was also applied to develop the Papyrus-RT runtime system. This system was originally developed in C++ with some non-object-oriented code. The experiment showed that the main difficulty of using our approach applied to such a system, was to reverse it into an exploitable UML model without loss of information. Once the processes of our synchronization solution were initialized, collaboration between software architects and programmers was made possible. We were then able to reap the benefits of MDE approaches and development was eased through automation brought by MDE. In particular, the maintainability and evolvability of the system was improved.

Realization-wise, in the future, we would like to make our implementation even more generic. Indeed, we had to modify tools like EMF Compare to suit the implementation of our solution with other Eclipse technologies. Currently we are also implementing our generic synchronization pattern for UML and Java.

Our work was in part motivated by the perceived gap between diagram-based languages and textual languages, which limited adoption of MDE. In the future, we would like to measure how our solution can help blur the perceived distinction between model and code, by the MDE community and traditional software communities.

References

[1] O. Acceleo. "Acceleo Homepage". https://www.eclipse.org/umlgen/, 2015.

[2] Altova. "UModel". http://www.altova.com/umodel.html, 2015.

[3] AndroMDA. AndroMDA Model Driven Architecture Framework. http://andromda.sourceforge.net/, 2016.

[4] U. Assmann. Automatic roundtrip engineering. *Electronic Notes in Theoretical Computer Science*, 82(5):33–41, 2003.

[5] G. Bergmann, I. Dávid, A. HegedÃijs, A. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró. VIATRA 3: a reactive model transformation platform. In *Theory and Practice of Model Transformations*, pages 101–110. Springer, 2015.

[6] G. Bergmann, I. Ráth, G. Varró, and D. Varró. *Change-driven model transformations*, volume 11. mar 2011.

[7] T. Booth and S. Stumpf. End-User Experiences of Visual and Textual Programming Environments for Arduino. In *End-User Development*, volume 7897, pages 25–39. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[8] N. C. C. Brown, M. Kolling, and A. Altadmri. Position paper: Lack of keyboard support cripples block-based programming. In *Proceedings of the 2015 Blocks and Beyond Workshop*, Atlanta, GA, USA, 2015.

[9] C. Brun and A. Pierantonio. Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008.

[10] I. Burnett, M. Baker, C. Bohus, P. Carlson, S. Yang, and P. Van Zee. Scaling up visual programming languages. *Computer*, 28(3):45–54, Mar, 1995.

[11] CEA. Papyrus-RT Website. https://www.eclipse.org/papyrus-rt/, 2016.

[12] A. Charfi, C. Mraidha, and P. Boulet. An Optimized Compilation of UML State Machines. In *Proceedings of 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, Guangdong, China, 2012.

[13] E. J. Chikofsky, J. H. Cross, and others. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1):13–17, 1990.

[14] S. Conversy. Unifying Textual and Visual: A Theoretical Account of the Visual Perception of Programming Languages. In *Proceedings of the 2014 ACM Internation Symposium on New Ideas, New Paradigs, and Reflections on Programming & Software*, Portland, OR, USA, 2014.

[15] D. Cutting and J. Noppen. An Extensible Benchmark and Tooling for Comparing Reverse Engineering Approaches. *International Journal on Advances in Software*, 8(1):115–124, 2015.

[16] R. Eramo, A. Pierantonio, J. Romero, and A. Vallecillo. Change Management in Multi-Viewpoint System Using ASP. In *Proceedings of the 12th Entreprise Distributed object Computing Conference Workshops*, Munich, Germany, Sep, 2008.

[17] M. Erwig and B. Meyer. Heterogeneous visual languages-integrating visual and textual programming. In *Proceedings of the 11th IEEE International Symposium on Visual Languages*, 1995.

[18] A. Finkelstein, J. Kramer, and M. Goedicke. ViewPoint Oriented Software Development. In *Proceedings of the 3rd International Workshop on Software Engineering and its Applications*, Toulouse, France, 1990.

[19] S. Fischer, Z. Hu, and H. Pacheco. GRACE TECHNICAL REPORTS âĂIJ Putback âĂİ is the Essence of Bidirectional Programming. (December), 2012.

[20] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.

[21] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.

[22] D. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[23] E. U. Generators. "Eclipse UML Generators Homepage". https://eclipse.org/acceleo/, 2015.

[24] S. Gérard. Once upon a Time, There Was PapyrusâĂę. In *Proceedings of 3rd International Conference on Model-Driven Engineering and Software Development*, Montreal, Canada, 2015.

[25] H. Giese and R. Wagner. Incremental Model Synchronization with Triple Graph Grammars. In *Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems*, Genova, Italy, 2006.

[26] H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In *Model Driven Engineering Languages and Systems*, pages 543–557. Springer, 2006.

[27] M. Goedicke, B. Enders, T. Meyer, and G. Taentzer. ViewPoint-Oriented Software Development: Tool Support for Integrating Multiple Perspectives by Distributed Graph Transformation. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785, pages 43–47. Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, Jun, 2000.

[28] M. R. C. Hafeez Osman. Correctness and Completeness of CASE Tools in Reverse Engineering Source Code into UML Model. *GSTF Journal on Computing*, 2(1):193–201, 2012.

[29] F. Hermann, H. Ehrig, C. Ermel, and F. Orejas. Concurrent model synchronization with conflict resolution based on triple graph grammars. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7212 LNCS, pages 178–193, 2012.

[30] T. Hettel, M. Lawley, and K. Raymond. Model Synchronisation: Definitions for Round-Trip Engineering. In *Theory and Practice of Model Transformations*, volume 5063, pages 31–45. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[31] T. Hettel, M. Lawley, and K. Raymond. Model synchronisation: Definitions for round-trip engineering. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5063 LNCS, pages 31–45, 2008.

[32] J. Hutchinson, M. Rouncefield, and J. Whittle. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, Honolulu, HI, USA, 2011.

[33] J. Hutchinson, J. Whittle, and M. Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161, Sept. 2014.

[34] IBM. Rational DOORS. http://www-03.ibm.com/software/products/en/ratidoor, 2016.

[35] IBM. Rational Rhapsody Website. http://www-03.ibm.com/software/products/en/ratirhapfami, 2016.

[36] S. Jörges. Construction and evolution of code generators: A model-driven and service-oriented approach. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7747:1–265, 2013.

[37] F. Jouault, F. Allilaire, J. BÃľzivin, and I. Kurtev. ATL: A model transformation tool. *Science of computer programming*, 72(1):31–39, 2008.

[38] A. Kazakova. Infographic: C/C++ facts we learned before going ahead with CLion. http://blog.jetbrains.com/clion/2015/07/infographics-cpp-facts-before-clion/, 2016.

[39] S. Kelly and J. P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. 2007.

[40] E. Kindler and R. Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. *University of Paderborn*, 2007.

[41] T. Klein, U. A. Nickel, J. Niere, and A. Zündorf. From uml to java and back again. Technical report, University of Paderborn, Paderborn, Germany, 1999.

[42] K. Kojima, Y. Matsuda, and S. Futatsugi. LIVE-Integrating visual and textual programming paradigms. In *Proceedings of the 1989 IEEE Workshop on Visual Languages*, Rome, Italy, 1989.

[43] A. Kusel, J. Etzlstorfer, E. Kapsammer, P. Langer, W. Retschitzegger, J. Schoenboeck, W. Schwinger, and M. Wimmer. A Survey on Incremental Model Transformation Approaches. In *Proceedings of the 7th Models and Evolution Workshop*, Miami, FL, USA, 2013.

[44] F. Lanubile, C. Ebert, R. Prikladnicki, and A. Vizcaino. Collaboration Tools for Global Software Engineering. *IEEE Software*, 27(2):52–55, Mar. Mar, 2010.

[45] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *Journal of Systems and Software*, 7(4):341–355, Dec, 1987.

[46] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, 39(6):869–891, Jun, 2013.

[47] K. Matsuda and M. Wang. Applicative bidirectional programming with lenses. *Proceedings of the 20th ACM SIGPLAN . . .*, pages 38–41, 2015.

[48] MODELS. Models conference. http://www.modelsconference.org, 2016.

[49] T. G. Moher, D. Mak, B. Blumenthal, and L. Levanthal. Comparing the comprehensibility of textual and graphical programs. In *Proceedings of the 5th workshop on Empirical Studies of Programmers*, Palo Alto, CA, USA, 1993.

[50] R. Navarro-Prieto and J. J. Canas. Are visual programming languages better? The role of imagery in program comprehension. *International Journal of Human-Computer Studies*, 54(6):799–829, Jun, 2001.

[51] I. No Magic. "magic draw". https://www.nomagic.com/products/magicdraw.html, 2016.

[52] OMG. Software & Systems Process Engineering Metamodel specification (SPEM) Version 2.0. http://www.omg.org/spec/SPEM/2.0/PDF, 2008.

[53] OMG. UML Specification. Specification 2.4.1, OMG, Aug. 2011.

[54] Q. Omg. Meta Object Facility (MOF) 2 . 0 Query / View / Transformation Specification. *Transformation*, (January):1–230, 2008.

[55] E. Paesschen, W. Meuter, and M. D'Hondt. *Model Driven Engineering Languages and Systems: 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005. Proceedings*, chapter Self-Sync: A Dynamic Round-Trip Engineering Environment, pages 633–647. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[56] V. Paradigm. "visual paradigm". http://www.visual-paradigm.com/, 2015.

[57] M. Petre. Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, Jun, 1995.

[58] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

[59] E. Schanzer, K. Shriram, and F. Kathi. Blocks Versus Text: Ongoing Lessons from Bootstrap. In *Proceedings of the 2015 Blocks and Beyond Workshop*, Atlanta, GA, USA, 2015.

[60] D. C. Schmidt. Model-driven engineering. *Computer*, 39(2):25, 2006.

[61] B. Selic. What will it take? A view on adoption of model-based methods in practice. *Software & Systems Modeling*, 11(4):513–526, Oct. 2012.

[62] S. Sendall and J. KÃijster. Taming model round-trip engineering. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development*, Vancouver, Canada, 2004.

[63] Y. Solutions. Uml lab. http://www.uml-lab.com, 2012.

[64] SparxSysemx. "enterprise architect". http://www.sparxsystems.com/products/ea/, 2016.

[65] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[66] P. Stevens. A landscape of bidirectional model transformations. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5235 LNCS, pages 408–424, 2008.

[67] Visure. Visure Requirements. http://www.visuresolutions.com/, 2016.

[68] A. Von Mayrhauser and A. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, Aug. 1995.

[69] K. Whitley. Visual Programming Languages and the Empirical Evidence For and Against. *Journal of Visual Languages and Computing*, 8(1):109–142, Feb, 1997.

[70] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal. Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem? In *Model-Driven Engineering Languages and Systems*, volume 8107, pages 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 2013.

[71] A. Wider. Towards combinators for bidirectional model transformations in scala. *Software Language Engineering*, 2012.

[72] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *Proceedings of the 22nd IEEE/ACM Internation Conference on Automated Software Engineering*, 2007.