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 increased automation to the 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 is increasing evidence that MDE facilitates 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. They consider such languages to be much more suitable for describing architectures compared to textual languages. Synchronizing manually written artifacts in either type of language is not simple and is often very time-consuming and error prone unless it is supported by automated methods and tools.

To solve this issue, we propose a methodological pattern for model-code synchronization supported by corresponding tooling. The solution tackles a fundamental problem of round-trip engineering: synchronization between concurrently evolving artifacts. On one side we have the architecture model maintained by the software architects, while on the other, we have code written by programmers. Applying our approach for the development of an actual 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 Modeling Tools

1. INTRODUCTION

Model-Driven Engineering (MDE) [35] is a development paradigm that seeks to increase productivity, by, among other things, increasing the level of automation. Traditionally, the MDE community draws a sharp distinction between development using a diagram-based language (commonly called a modeling language in the MODELS [28] community) and development using a textual language (i.e. a programming language or code). Proponents of MDE have argued that adopting model-centric approaches would benefit software developers due to its many advantages [35], especially for development of complex systems. They encourage the greater use of model-driven development, by, among other initiatives, providing diagram-based languages supported by appropriate tools [14] such as automated code generation.

In industrial practice, there is still significant reticence to adopt a fully model-centric approach [19, 35]. The reticence is due in part to the perceived gap [2] between diagram-based languages and textual languages. On one hand, programmers often shy away from graphical languages, preferring to use the more familiar combination of a programming language and Integrated Development Environment (IDE). On the other hand, software architects, working at higher levels of abstraction, tend to favor the use of models, and therefore prefer graphical languages for describing the architecture of the system.

Even in teams who embrace a model-driven approach, the gap between model and code is not bridged seamlessly. The survey described in [20] polled stakeholders in companies who use MDE approaches. It notes that 70% of the respondents primarily work with models, but still require manually-written code to be integrated. Furthermore, 35% of the respondents answered that they spend a lot of time and effort synchronizing model and code.

The back-and-forth switching between diagram-based languages and textual languages, as well as between model and code, has hindered the adoption of MDE in industrial practice. To solve this issue, we believe the sharp distinction between model and code must be blurred. We feel that a collaborative solution must be offered to different categories of developers (e.g. architects, programmers), who use different development practices. The solution must not impose a choice between a diagram-based language and a textual language.

Collaboration between developers producing different types of artifacts, in different languages, using different tools, raises the issue of artifact synchronization. This is a well-known concern with round-trip engineering. Namely, round-trip engineering requires the ability to maintain consistency between multiple concurrently evolving software artifacts [36]. It is related to traditional software engineering disciplines such as forward and reverse engineering [4]. Round-trip engineering achieves synchronization between related artifacts that may evolve concurrently by incrementally updating each artifact to reflect editions made in the other artifacts.

The work presented in this paper is motivated by the open-source Papyrus-RT [3] runtime system which was developed in collaboration by companies A, B, and C. These companies are contributors to the MDE eco-system. The architecture of this system follows an object-oriented paradigm, and it was initially implemented as a C++ project with about 15,000 lines of code. This project encountered the problem of collaboration between MDE developers and programmers working in the traditional manner.

In this paper, we propose to use automated model-code synchronization as a means of bridging the gap between models edited by software architects, and code written by programmers. Our approach is based on the following contributions, described in detail 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 co-evolve. This contribution is based on the following requirement and proposition:
   - Requirement: the availability of a generic IDE with functionalities necessary for model-code synchronization. The functionalities are not dependent of a particular approach or technology. The required IDE will be described in this paper.
   - Proposition: Processes to use the IDE to synchronize model and code based on several defined scenarios, which correspond to common development practices.
2. An Eclipse-based implementation of the approach for synchronization between UML models and corresponding C++ code.
3. Experience on applying the proposed solution for the development of a real-world 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. The tooling of our solution focuses on the Unified Modeling Language (UML), because it is a very widely used software architecture description language [26]. We want to synchronize models specified in a diagram-based language like UML with code in a popular textual programming language like C++ (used by 4.4 million [24] developers worldwide).

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. The approach is 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. COLLABORATING ACTORS AND USE-CASES OF SYNCHRONIZATION

In this section we define the actors who will use our model-code synchronization approach to collaborate during development. Then we define the main capabilities, as 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 Collaborating actors and development artifacts

In this paper we propose a methodological model-code synchronization pattern for collaboration between software architects and programmers. For the sake of generality, we postulate that the architect and programmer are actors with starkly opposite development practices. This allows the approach to be used even in 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 code implementation for the latter. First, we introduce the concepts of *development artifact* and *baseline artifact*.

Definition 1 (Development artifact). *A development artifact is a artifact, as defined in [30], 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 but part of the implementation. For example a model can be used for implementation 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 in this paper 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 a process. Manual edition of artifacts other than the baseline artifact is forbidden.*

Two primary actors, called *model-driven developer* and *code-driven developer*, are introduced. The main difference between them is what they consider as the baseline artifact.

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

In other words, for the model-driven developer only the model should be edited manually. The code must always be produced from the model automatically by some process that guarantees that the code is consistent with the model.

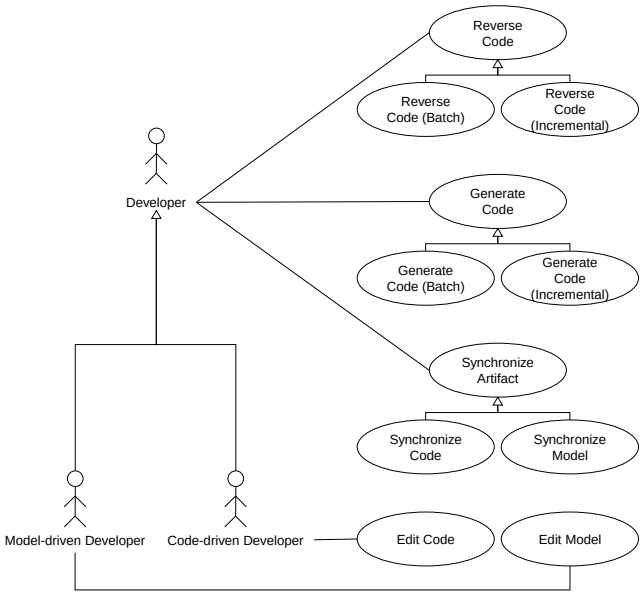

Figure 1: Use-cases of IDE for model/code edition and synchronization

A software architect is a kind of the model-driven developer who edits the model to specify the architecture of the system. An architect presumes that the reference for the architecture of the system should be specified as a model.

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

A programmer is a specialization of the code-driven developer. Indeed, programmers may modify the code, such as editing method bodies. The code is then the main reference for the implementation of methods.

2.2 Main use-cases of IDE for collaboration between developers

In this section we propose a generic IDE with the main 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 implies that 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 the artifact is the model or code.

There are also some use-cases related to the synchronization of artifacts. The `Synchronize Artifact` use-case is the synchronization of two artifacts by: (1) comparing them, (2) updating each artifact with editions made in the related artifact, and (3) reconciling conflicts when appropriate. The `Synchronize Model` and `Synchronize Code` use-cases are specializations where, respectively, the model or the code are the artifacts being synchronized.

The `Generate Code` use-case is related to forward engineering. 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 [15]). *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 [15] as model transformation that does not generate the whole target model from scratch but only updates the target model by propagating editions made in the source model.

Derived 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 an edited model, and existing code, and then updating the code by propagating editions in the model to the code.*

Finally, the `Reverse Code` use-case is related to reverse engineering. `Reverse Code` is the production of a model, in a modeling language, from code, written in a programming language. The developer can either use `Reverse Code (Batch)` or `Reverse Code (Incremental)`, which are 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. The 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 updating the model by propagating editions in the code to the model.*

For readability, in this paper we will sometimes 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. 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 best.

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

3. PROCESSES TO SYNCHRONIZE MODEL AND CODE

This section describes some processes for synchronizing model and code in different scenarios. The scenarios are differentiated according to the type of developers, i.e. the type of actors, working on the system to be developed.

For each of the scenarios, we assume that the common starting point of each process is a complete model. Therefore, if there is some legacy code sitting outside of the model, it must first be reversed back before the processes described below can be applied. This can be done with the `Reverse Code (Batch)` use case of the IDE proposed in Section 2.2.

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

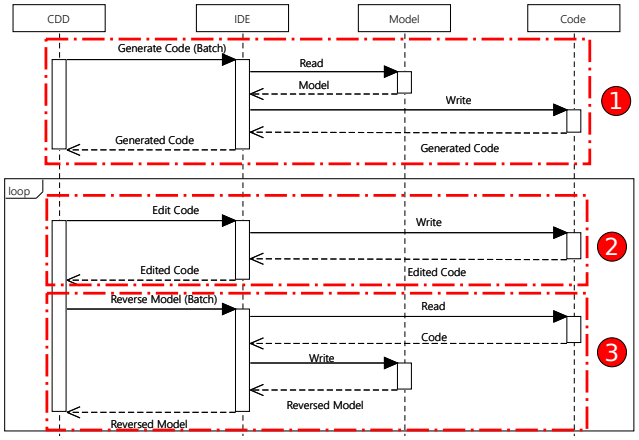

Figure 2: Synchronization process for scenario 1, in which only code is edited (CDD = Code-Driven Developer). The API calls for Model and Code are represented generically as "Read" and "Write".

3.1 Scenario 1: code-only editions

The assumption in scenario 1 is that the code is the only baseline artifact. (In other words, only the code may be edited manually.) We assume that during development, the model needs to be synchronized sporadically with the code. This scenario is usually more appropriate for the code-driven developer.

Figure 2 shows a sequence diagram that illustrates the process for synchronizing code and model in scenario 1. The general approach is to always overwrite the model with a new model produced from the edited code.

In Figure 2, the code-driven developer interacts with the IDE which then writes and reads from code and model. Keep in mind the general assumption that the starting point of the process is a model. Note that messages between the code-driven developer and the IDE are the use-cases of the IDE proposed in Section 2.

In Figure 2, there are some markups to highlight the general stepsof the process. The general steps of the process are described as follows:

- Scenario 1 synchronization process general steps -

Step 1 To begin implementation, the code is generated in batch mode from the model. 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 change cycle, the code can be reversed. Therefore steps **(2)** and **(3)** may be repeated multiple times in this process.

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

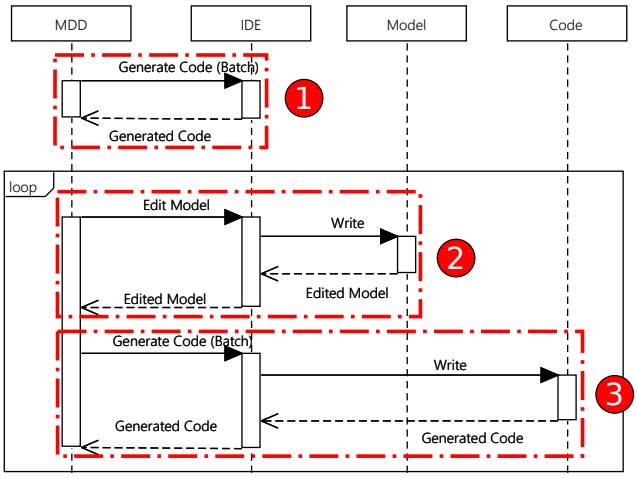

Figure 3: Synchronization process for scenario 2, in which only model is edited (MDD = Model-Driven Developer). The API calls for Model and Code are represented generically as "Read" and "Write".

3.2 Scenario 2: model-only editions

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

Figure 3 shows a sequence diagram that illustrates 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.

Highlighted steps of the process shown in Figure 3 are described as follows:

- Scenario 2 synchronization process general steps -

Step 1 Code is generated in batch mode from the model. (To simplify Figure 3, we do not show the Read and Write interactions for this step, as was the case in Figure 2.)

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 fully-fledged round-trip engineering with a need for 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 additional synchronization strategies, since only either the code or the model is the

baseline artifact, i.e. only one is edited manually and there are no concurrent editions being made to the two artifacts. Therefore batch code generation and reverse are sufficient for synchronization.

However, in scenario 3, there is no unique baseline artifact; both the model and the code may be edited manually. Therefore they may evolve concurrently during development activities and synchronization issues will occur. This scenario tackles the problem where model-driven and code-driven developers are working together on the same system.

We propose two synchronization strategies for this scenario. The general approach behind our synchronization strategies is to represent one artifact in the language of its corresponding other artifact. These two can then be compared. For this, we define a concept of a *synchronization artifact*:

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

For example, a synchronization artifact can be code that was generated from the edited model in batch mode. In that 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 in which the synchronization artifact is code, and the other in which the synchronization artifact is a model. We propose two strategies so the developer can choose to either use the `Synchronize Code` or `Synchronize Model` use-cases of the IDE. The choice may be determined by preferred development practices or the availability of suitable tools (e.g. the programmer may prefer to synchronize two artifacts, both represented in the same programming language, because he prefers to work exclusively with code).

Figure 4 shows the first synchronization strategy based on using code as the synchronization artifact. The highlighted general steps of the process shown in Figure 4 are described as follows:

- Scenario 3 synchronization process steps -

Step 1 Both the model and code may be edited concurrently. (To simplify Figure 4, we don't show the Read and Write interactions for this step.) After both artifacts have been edited concurrently, we need to synchronize them.

Step 2 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 3 The synchronization artifact is synchronized with the edited code. Since the synchronization artifact is code itself, this step is done with the `Synchronize Code` use-case of the IDE.

Step 4 Once synchronization artifact and edited code are synchronized, the former is reversed incrementally to update the edited model.

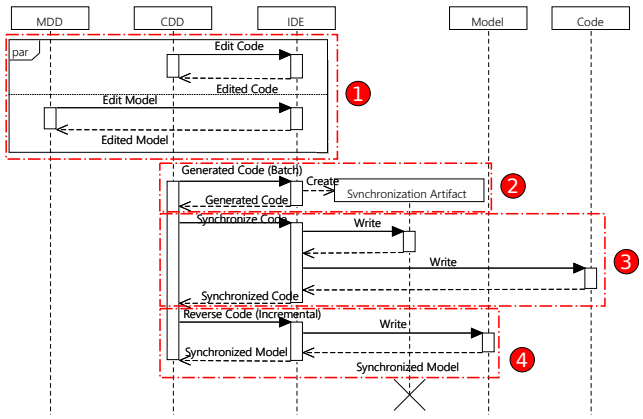

Figure 4: Synchronization process for scenario 3, in which the model and the code are concurrently edited with code as the synchronization artifact (CDD = Code-Driven Developer, MDD = Model-Driven Developer). The API calls for Model and Code are represented generically as "Read" and "Write".

The second strategy, based on using model as the synchronization artifact, is the opposite of the first strategy. In the second strategy, the synchronization artifact is obtained by reversing the edited code in batch mode. Afterwards the synchronization artifact is synchronized with the edited model. Finally, we generate code incrementally from the synchronization artifact to update the edited code.

We propose two strategies based on the preferences of the developers. They may even use both strategies, successively, as a kind of hybrid strategy. This may be useful when developers want to synchronize parts of the system using one strategy, and other parts using the other strategy. For example, they may choose to synchronize method bodies using strategy 1, where the synchronization artifact is code. Then strategy 2, in which the synchronization artifact is a model, is used to to synchronize architectural elements of the system.

In the next section we propose an implementation of an IDE and the proposed synchronization processes.

4. AN IMPLEMENTATION: SYNCHRONIZING UML MODELS AND C++ CODE

The proposed model-code synchronization approach can be automated. We examine an Eclipse-based implementation of the IDE proposed in Section 2.2. The implementation is used in the synchronization processes proposed in Section 3. Our implementation targets synchronization of Object Management Group (OMG) standard UML 2 and C++11 code.

The use-cases of the IDE are implemented with some Eclipse technologies. More information on the technologies is available on the Eclipse Projects website [8].

Eclipse CDT is an IDE for C++ development. It is used in the `Edit Code` use-case. Eclipse **Papyrus** is used for the `Edit Model` use-case. Papyrus is an open-source UML modeler that uses the Eclipse Modeling Framework (EMF) implementation of the OMG-standard UML 2. Papyrus supports UML profiles for domain-specific modeling, and we 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 do 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 [25]. The **Viatra** 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 is used for the `Synchronize Model` use-case when models are implemented with EMF. We adapted EMF Compare in order to synchronize UML models for our specific work. Eclipse CDT is used for the `Synchronize Code` use-case with its built-in C++ features.

The next section describes some experiments performed using the above implementation of our synchronization solution.

5. EXPERIMENTS AND EVALUATIONS

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 realistic system. In the following subsections, we first describe the results of some simulations designed to test the proposed approach. Next, we report on a realistic case-study intended to help us assess certain scalability and usability aspects.

5.1 Simulations to assess synchronization processes

Simulations have been conducted to test that the implementation of our proposition respects the round-trip engineering laws [12] *right-invertibility* and *left-invertibility*. These laws are stated as follows:

Law 1: Right-invertibility means that not editing the code (or model, respectively) shall be reflected as not editing the model (respectively code). If no editions are made to the code, the model used for generating the code and the model received by immediately reversing the generated 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 fully recreated by applying code generation to the updated model.

In the following sections, the model generator used for the simulations is presented first. Then, the simulations performed for both above laws are described. Finally, we discuss the simulation of the process defined for scenario 3 (Section 3.3), in which both the model and the code are edited concurrently.

5.1.1 Randomly generated UML models

In the simulations, random UML models are generated with a configurable model generator. The generated models contain C++ features represented via UML. The generator can be configured to generate a desired average number of each type of C++ feature to be represented as a UML element.

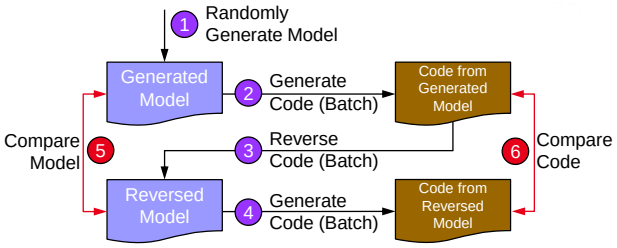

Figure 5: Simulation 1 for Law 1 (right-invertibility)

Three packages are generated for each model. Each package contains 60 classes, and, on the 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 of the appropriate 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.

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 5.

The general idea behind the simulation procedure is to do a full round-trip of the 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 model is compared with code generated from the reversed model in step **(6)**.

Code generation and reverse 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. If these numbers match, EMF Compare is used. Comparison of code is done by using the built-in code comparison tool in the 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. We limited ourselves to 200 models for practical reasons. These same models were later used for a simulation that involved some manual editions (presented in the next subsection).

After launching the simulation for the 200 models, no differences were found during model comparison and code comparison. This result assesses that the tooling of our model-code synchronization approach with respect to Law 1 of round-trip engineering.

5.1.3 Simulation for Law 2 left-invertibility

In the second simulation, Law 2 was evaluated. The pro-

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

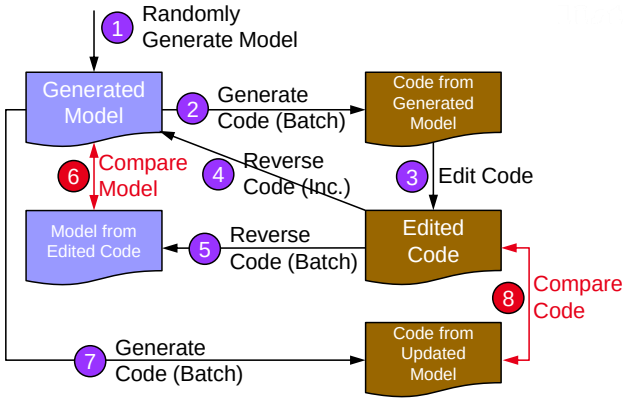

Figure 6: Simulation 2 for Law 2 (left-invertibility)

cedure for the simulation is shown in Figure 6.

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 6, the procedure to test our tooling consisted 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 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** (the 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**.

Table 2: Editions and the modifications events they trigger (A = `ADDED`, C = `CHANGED`, R = `REMOVED`)

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
Manual editions	39	34	30	No	No

The simulation is run for 200 randomly generated models. We limited ourselves to 200 models because in step **(3)**, some of the editions have to be done manually to emulate real development conditions. Therefore the limit of 200 models is purely based on practical concerns.

During the simulations, the code generated from each model undergoes seven kinds of editions independently (the kinds of editions performed are listed in Table 2). As its name suggests, only "Manual editions" were done manually by a developer, to emulate actual development conditions.

The editions triggered three kinds of Eclipse CDT modification events: `ADDED`, `REMOVED`, and `CHANGED`. Events `ADDED` and `REMOVED` mean addition and deletion of classes, attributes or methods to the code. Event `CHANGED` is the update of elements in code including (1) renaming attributes, classes, or methods, (2) changing the type of attributes, parameters, or (3) changing the behavior of methods.

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

For all 200 models, no differences were found during model comparison and code comparison. This result assesses that the tooling of our model-code synchronization approach with respect to Law 2 of round-trip engineering.

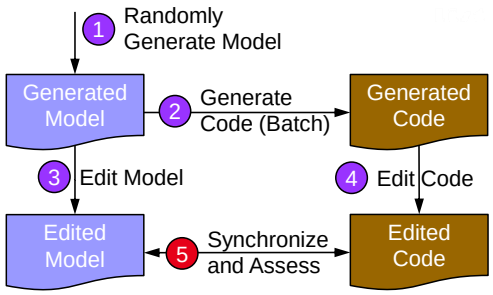

Figure 7: Simulation 3 for concurrent editions case

5.1.4 *Simulation for concurrent edition of model and code*

A third simulation aims at emulating the process described in Figure 4 of Section 3.3, in which the model and the code are concurrently edited. A simplified representation of the procedure of simulation 3 is shown in Figure 7. The simulation is executed for 200 randomly generated UML models and their generated C++ code.

The idea behind the procedure in Figure 7 is to simulate editions in the original randomly generated model, and its corresponding generated code (step **(1)** to **(4)**). To simplify the simulation, the simulator only introduces attribute renaming model-side. On the code-side, the simulator only makes editions to method bodies. Edited model and code must then be synchronized and then we must assess that they are indeed synchronized. This is done in step **(5)**.

To synchronize edited model and edited code, first we use the strategy based on code as the synchronization artifact. The simulator propagates all method body editions from the edited code to the synchronization artifact. The simulator then propagates all attribute renaming from the synchronization artifact to the edited code. Next, the synchronization artifact is reversed incrementally to produce the edited model. At this point the model and the code are considered synchronized.

To assess that both synchronized model and code are images of one another, the synchronized code was reversed in batch mode to a new model. The new model was then compared with the synchronized model. No differences were found during these simulations. We also generated new code in batch mode from the synchronized model. The new code was compared to the synchronized code. No differences were found during the comparison.

The same simulation is repeated for the case where the model is used as synchronization artifact. Again the synchronized code and model were images of one another.

Once the tooling of the proposed model-code synchronization approach was verified through the three simulations, we decided to apply it to the development of a practical real-world system. The results of this experiment are discussed in the next section.

5.2 Papyrus-RT runtime case-study

In Section 5.1 the approach was tested for UML and C++ synchronization, using simulations. In this section we describe the application of the full synchronization approach to a case-study. The intent here was to evaluate:

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

Table 3: Differences in Papyrus-RT runtime versions

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 outside of class	Refactored as entities inside a class. Attributes and functions are static with visibility defined according to scope of original variables and functions.

The case-study is related to the development of the runtime underlying Papyrus-RT [3]. This latter is an open-source custom modeling tool, based on Papyrus, that supports UML extensions for the design of a category of event-driven real-time systems (UML-RT). Papyrus-RT features full automated code generation for UML models. The generated code executes within a corresponding runtime environment, which provides C++ realizations of the high-level concepts defined in UML-RT.

At the beginning of the project, the resource available to develop the aforementioned runtime was an experienced C++ programmer. For the 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 satisfactory, it would be beneficial to exploit the advantages that MDE provides. Thus, an MDE approach would improve both the maintainability and the evolvability of the runtime. Moreover, as development progressed, the project was expanded to include new developers, who are proponents of MDE and are eager to work with models.

As a result, this project represented an ideal case study to assess the practicality of the proposed approach. In particular, we were interested in determining its usability and 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 14,945 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 original runtime contained some elements that were not object-oriented. Consequently, some modifications and refactorings were made to the original runtime to make it fully object-oriented. This enabled it to be modeled entirely in a language like UML. Table 3 shows the two main differences between the original runtime, and the revised object-oriented runtime.

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 22,053 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 unit 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 the proposed approach introduced MDE style development for the Papyrus-RT runtime. This brought about several advantages as several manual tasks were automated:

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

In applying the proposed approach to achieve a collaborative development of the runtime system, we found that the only real difficulty faced was initializing the synchronization processes. This required the reverse engineering of the original runtime, with some non-object-oriented code, into an object-oriented UML model. The reverse was successful following some modifications and refactorings. Once the full model-code synchronization process was in place, we were able to use it and develop concurrently both the UML model and its generated C++ code.

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 desire to reduce the gap between model and code, between diagram-based languages and textual languages. We use automation-supported model-code synchronization as a means to achieve this goal. In the following sections, we compare our propositions to related works recorded in the literature.

6.1 Comparison of diagram-based and textual languages

Diagram-based languages, also called graphical languages, are a subset of visual languages,. A number of efforts were undertaken that compared textual languages and visual languages.

Many works [32, 34] discuss the pros and cons of visual languages, compared to textual languages, in various contexts. Others [2] 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. Instead, we prefer to blur the boundaries between them. The idea that visual languages can co-exist with textual languages has been noted by other authors.

In [5] the authors argue that the gap between textual and visual languages is narrow. They propose a framework to represent code in a visual language to improve comprehension of the code.

In [11] the authors propose a generic framework to use both visual and textual languages at the same time. The visual artifacts are translated to textual code. Much importance is given to the human factor, i.e. the framework can be customized based on developer preferences.

In domains such as requirements engineering, tools, such as IBM Rational Doors [21] and Visure Requirements [38], 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.

However, none of these systems are intended to support full implementation of a system. Generally, one of the artifact is only used to assist comprehension. There is no need for synchronization between artifacts because the transformation only proceeds in one direction.

Our argument that, in order to be more efficient, developers should not be forced to choose between a diagram-based or textual language, is directly related to software comprehension. Works [39] 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 specifically both software architects and programmers. The former generally foster diagram-based languages for architecture description and comprehension. The latter prefer textual languages since they are deemed much more expressive for specifying fine-grained algorithms, 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, viewpoint synchronization, and model synchronization.

Round-trip engineering of model and code.

Several commercial and open-source tools [9, 22, 29, 37] support round-trip engineering between UML models and code. Systematic reviews of some of these tools are available in [6]. Usually these tools only support architectural elements on the 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 [37] 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 since there are no concurrent changes, which limits their applicability. Finally, some tools [9] do not support a real incremental reverse or code generation; instead, they treat change (e.g. renaming) as deletion followed by addition.

Some round-trip engineering techniques restrict the development artifact to avoid synchronization problems. Partial round-trip engineering and protected regions are introduced in [7]. Such techniques aim to preserve code editions which cannot be propagated to models. This approach separates the code regions that are generated from models from regions which are allowed to be edited by developers. In contrast to our work, this form of round-trip engineering is unidirectional and does not support iterative development [23]. Furthermore we do not restrict editions on model and code.

Viewpoint synchronization.

Both models and code can be considered simply as different viewpoints of the same system. Viewpoints enable the partitioning of the model of a system into several representations. Synchronization between viewpoints is crucial to maintain their consistency.

In [10] the authors improve the modeling of relationships and constraints between elements in different viewpoints in order to better guarantee the consistency of viewpoints. In [17] the authors argue that inconsistencies will exist in systems developed with different actors, using different viewpoints. They suggest that tools must be able to tolerate inconsistencies. A distributed graph transformation is proposed to deal with the problem of formalizing the integration of multiple viewpoints in software development. Their work focuses on requirements engineering. In contrast, our approach targets specifically both model and code. Code is not usually considered in viewpoint synchronization because code is deemed to be too fine-grained. 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.

Many model synchronization techniques require the explicit mapping of source model and target model. The authors in [31] propose an injective mapping of elements in the source model to the target model. The mapping can be used for synchronization. Techniques and technologies, such as Triple Graph Grammar (TGG) [16], and QVT-Relation [33], allow synchronization between source and target elements that have non-injective mappings. The authors in [18] formalize TGG for synchronization of models that are concurrently edited. All of these techniques require a mapping model to connect the source and target models with typed traceability links, which need to be persisted in a model store [1]. This means that editing one model requires the presence of the other. Our model-code synchronization approach does not require a mapping model and an artifact may be edited independently of the presence of the other corresponding artifact.

Other techniques [13] are based on bi-directional transformations, which comprise a forward transformation of source to target model, and a backward transformation of target to source model. Bi-directional transformations provides a novel mechanism for synchronization. Indeed, some works [27] derive a backward transformation based on forward transformation. However, such works do not offer any means to synchronize models that are concurrently edited.

A few approaches derive model synchronization from model transformation while allowing concurrent editions of both source and target models. In [40] the authors propose to automatically derive model synchronization of a source and a target model related by an ATL [8] model transformation. The synchronization is based on differentiating source and target model states. But, reflectable addition of an element in the target model is not well handled according to [40]. Our approach is generic and does not depend on a specific 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, such 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

This paper presents a generic model-code synchronization methodological pattern. The proposed approach is designed to support a continuous collaboration between software architects and programmers allowing each to use the language and tools of choice An Eclipse-based implementation of the approach was presented for synchronizing models based on the OMG-standard UML modeling language and the C++11 programming language.

Several simulations were used to validate our synchronization approach with respect to both laws of round-trip engineering. In addition, a simulation of scenarios in which both the model and the code were edited concurrently was performed, further demonstrating the viability of the approach, even in such a highly dynamic scenario.

The approach was then applied to a real-world application: 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 the need to edit and refactor the program code in such way that it can be reverse engineered into a corresponding UML model without loss of information. Once the processes of our synchronization solution were bootstrapped in this way, dynamic collaboration between software architects and programmers was possible. We were then able to reap the important benefits of MDE; development was facilitated through increased automation and system maintainability and evolvability were improved.

Currently we are implementing our generic synchronization pattern for several other programming languages, like Java. Part of our future work is to federate implementations of the system, in heterogeneous programming languages, as one or several models based on a common core in UML.

Our work was in part motivated by the perceived gap between diagram-based languages and textual languages, which impeded greater adoption of MDE among practitioners. Therefore, as a future work item, we would like to assess to what extent our solution helps in eliminating the unnecessary and dogmatic separation of models from code. We feel that this would be useful to both the MDE community and the more traditionally-oriented software communities. As MDE is gradually integrated into industrial practices, correct-by-construction approaches, through MDE, can be adopted to deliver higher quality software.

References

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

[2] 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.

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

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

[5] 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.

[6] 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.

[7] K. Czarnecki, M. Antkiewicz, and C. H. P. Kim. Multi-level customization in application engineering. *Communications of the ACM*, 49(12):60, Dec. 2006.

[8] Eclipse Foundation. Eclipse Projects. https://www.eclipse.org/projects/.

[9] Eclipse UML Generators. Eclipse UML Generators Homepage. https://eclipse.org/acceleo/, 2015.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

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

[29] No Magic, Inc. Magic Draw. https://www.nomagic.com/products/magicdraw.html, 2016.

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

[31] 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.

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

[33] QVT OMG. Meta Object Facility (MOF) 2.0 Query / View / Transformation Specification. *Transformation*, pages 1–230, 2008.

[34] 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.

[35] 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.

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

[37] SparxSystem. Enterprise Architect. http://www.sparxsystems.com/products/ea/, 2016.

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

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

[40] 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.