Stepwise Adoption of Continuous Delivery in Model-Driven Engineering

Jokin Garcia[1] and Jordi Cabot[2]

[1] IK4-IKERLAN, Arrasate, Spain
`jgarcia@ikerlan.es`
[2] ICREA-UOC, Barcelona, Spain
`jordi.cabot@icrea.cat`

Abstract. *Continuous Delivery* (CD) and, in general, *Continuous Software Engineering* (CSE) is becoming the norm. Still, current practices and available integration platforms are too code-oriented. They are not well adapted to work with other, non text-based, software artifacts typically produced during early phases of the software engineering lifecycle. This is especially problematic for teams adopting a *Model-Driven Engineering* (MDE) approach to software development where several (meta)models (and model transformations) are built and executed as part of the development process. Typically, (part of) the code is automatically generated from such models. Therefore, in a complete CD process, changes in a model should trigger changes on the generated code when appropriate.
A step further would be to apply CD practices to the development of modeling artefacts themselves. Analogously to "traditional" CD, where the goal is to have the mainline codebase always in a deployable state, the aim would be to have the modeling infrastructure always ready to be used. Those models could be the final product themselves or an intermediate artifact in a complete CSE process as described above.
Either way, a tighter integration between CD and MDE would benefit software practitioners by providing them with complete CSE, covering also analysis and design stages of the process.

Keywords: Continuous Evolution, Continuous Delivery, Model-Driven Engineering

1 Introduction

Gone are the days when developing projects required a mere compiler. Nowadays, software engineering is much more complex and heterogeneous, often involving several stacks, languages, and frameworks.

Software building tools have evolved accordingly and we have gone from *make* to *Gradle*, passing through *Ant* and *Maven*. Besides, agile practices and specifically *Continuous Delivery* (CD) has encouraged a more frequent software integration and testing. This philosophy of faster release cycles has expanded to the organizational level (e.g. for a rapid time-to-market and quality feedback) in what it is known as *Continuous Software Engineering* (CSE).

Unfortunately, so far this trend has left aside another parallel trend in software engineering: *Model-Driven Engineering* (MDE) [5]. MDE advocates for the rigorous use of models as key artifacts in all software engineering activities. Though this idea is far from new and we are still learning how to best effectively apply it in practice, recent studies suggest an increasing uptake of MDE and a more widespread use than commonly believed [22], specially when taking MDE on a broad sense (e.g. models beyond code-generation approaches, for instance, models used for communication purposes or software documentation).

Therefore, it is evident that software models play some role in most software development projects. Challenges for MDE adoption include social and organizational factors but also tool-related ones [23] such as synchronization problems between models and code. Clearly, a tighter integration between CD and MDE would benefit software practitioners by providing them with a more complete CSE, covering also analysis and design stages of the process. This integration is what we call Continuous Model-Driven Engineering.

This paper will be looking at this integration at two different levels. First, it will discuss how to add modeling artifacts as standalone executable components in a standard CD pipeline aimed at releasing a new software version.

Then, it will cover a more complex scenario where the target of the CD is a MDE artifact itself built as the result of a collaboration in a MDE ecosystem (a.k.a. megamodel in literature). Indeed, in many projects, the "modeling side" is a combination of models (possibly conforming to different metamodels, where each metamodel defines the possible set of well-formed models to be created with that language, similar to the relationship between programs and language grammars), model-to-model and model-to-text transformations (for the former, input and output are models, for the latter, the output is a text file, e.g. a piece of code generated from a model). Transformations can also be regarded as models on their own and conform to specific model transformation languages. As for the modeling languages, projects usually combine general modeling languages like UML with several *Domain-Specific Languages* (DSLs). DSLs can be reused from other projects or be developed adhoc for the current one, which implies creating their abstract syntax (grammar) and concrete syntax (notation) as part of the project itself.

Figure 1 tries to sketch how these elements relate to each other. As shown in Figure 1, model *Ma* is transformed into model *Mb* using a M2M transformation, and then model *Mb* is transformed into code through a M2T transformation. As can be seen from the number of relationships, CD of a MDE artifact is a complex task where changes on one artifact can trigger changes on several others that need to co-evolve together. This process requires tools for model comparison, merging, testing,... that react accordingly to (meta)model changes. While specific couples of on-demand evolution scenarios have been studied (metamodels-models [8], metamodels-transformations [12] and metamodels-editors [9]) no holistic and global approach has been proposed so far.

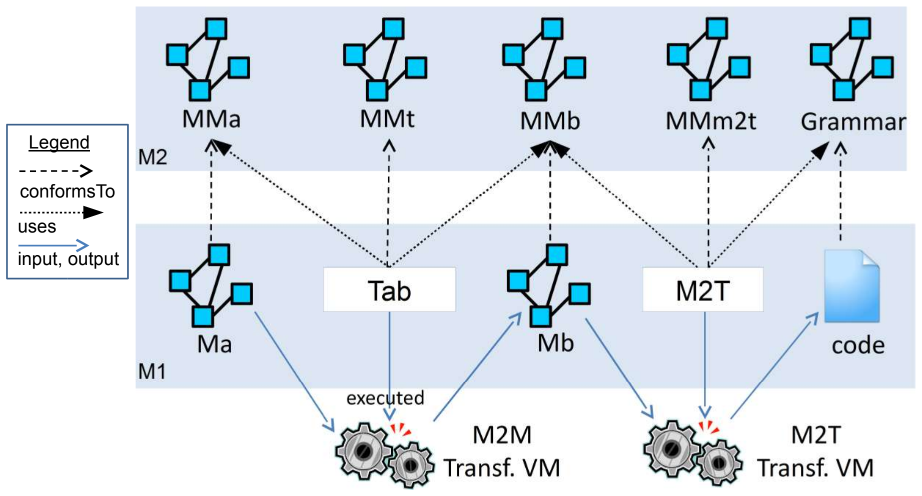


Fig. 1. MDE architecture

The remainder of the paper is structured as follows. Section 2 tries to clarify the glossary of terms that the mixture of Continuous* and Model-Driven domains entails. In Section 3 we analyze how can models can be integrated in CD processes and tools. Then, in Section 4 it is proposed to use CD practices to manage the evolution of MDE artifacts. Section 5 exposes related work; and finally, in Section 6 we conclude with a summary and future challenges.

2 Background

The Agile Manifesto was born in 2001. This manifesto claimed four values and twelve principles. The values are well-known: individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation and responding to change over following a plan. The principles are as well described in the manifesto [4].

In this agile context, and opposite to what it could be thought, teams are more likely to model than in traditional methodologies [23], as modeling supports many of their principles, as communication, rapid feedback or quality. MDE is a paradigm that uses models to develop software. Models conform to metamodels, and are transformed to other models or to code, building an ecosystem of related artifacts (Figure 1). These models can be used in workflows where they can be: validated, merged, compared, transformed, etc... [18]. These tasks are not in solitary confinement: they need to be integrated in heterogeneous projects managed with CD methodologies.

Rooted in the spirit of the manifesto, the Agile Model Driven Development (AMDD) method was conceived to ensure the emergence of effective architectures, requirements and designs. As the name suggests, AMDD is an agile version of MDD where the created models are not extensive, just good enough for the development cycle at hand. In opposition to the waterfall methodology where the modeling is done only in the beginning, in the agile software development lifecycle there are many cycles and in each of them modeling is present at the beginning [2].

The last step is introducing as well agility in the release step. Continuous Delivery is a subset of agile that emphasizes the need for software to be always ready for release. Contrary to the waterfall model that releases the software once all the functionality is developed, agile releases partial functionality throughout the development. In order to achieve this always-ready release philosophy, some techniques (e.g. test automation) and tools (e.g. Jenkins) are used.

To clarify these acronyms, we will adhere to the following reference terminology [15] : *Continuous Integration (CI)* is the frequent integration of code by all the members of a project. Build and tests are accomplished automatically in order to detect integration errors as soon as possible. *Continuous Delivery (CD)* is an extension of CI where it is guaranteed that the mainline is always in a deployable state, and that this deployment can be done in "one click". Opposed to CD where the deployment is manual, in *Continuous Deployment*, every time there is a commit, the software is automatically deployed to production. Continuous Software Engineering (CSE) is the organizational and cultural attempt to connect development with business strategy. All these practices are encompassed in what is known as Continuous*. In this paper we are going to use the term Continuous Delivery (CD), as it is closest to our proposal of always keeping the MDE infrastructure ready to be executed.

In order to achieve this automatic deployment, these techniques are based on the automation of the build. Specifically, a *deployment pipeline* divides and executes automatically different stages of the build; which are generally compilation, tests and deployment. This stages are, as well, broken up into jobs. This pipeline provides visibility of the whole process.

3 Integration of MDE tools in CD

In most software development projects, there is some degree of model use [22]. MDE components must collaborate with each other but also interface with other non-MDE tools, including CD servers [19], in a global CSE context.

This section looks at whether this integration is possible, focusing on the basic scenario of individual MDE artifacts used as part of a larger software development CD scenario. The key requirement of CD servers like Jenkins[3] is that an IDE cannot be used to build the software, as it does not guarantee a repeatable build. To be part of a CD pipeline, MDE tools must be able to be wrapped as jobs to be executed standalone, i.e. without human intervention, when called by the CD server.

Therefore, at its simplest level, integration of MDE in CD will be possible if we find at least one MDE tool, for each major MDE activity, offering some

[3] https://jenkins-ci.org/

kind of external interface (via API or shell access) that allows its integration in CD pipeline. And indeed, we do. Table 1 lists examples of such tools for each activity.

Modeling task	**Tool example**
Modeling framework	EMF[4]
Model to model transformation	ATL, Epsilon ETL, QVT
Model to text transformation	Acceleo, Epsilon EGL, Xpand
Model comparison	EMFCompare, Epsilon ECL
Model weaving/composition/merge	AMW, Epsilon EML
Model injection/extraction	Xtext, EMFText
Model validation	EMFtoCSP, Epsilon EVL

Table 1. Example of available tools for each modeling task

This alone is powerful enough to build CD pipelines for (model-based) development projects.

Illustrating example

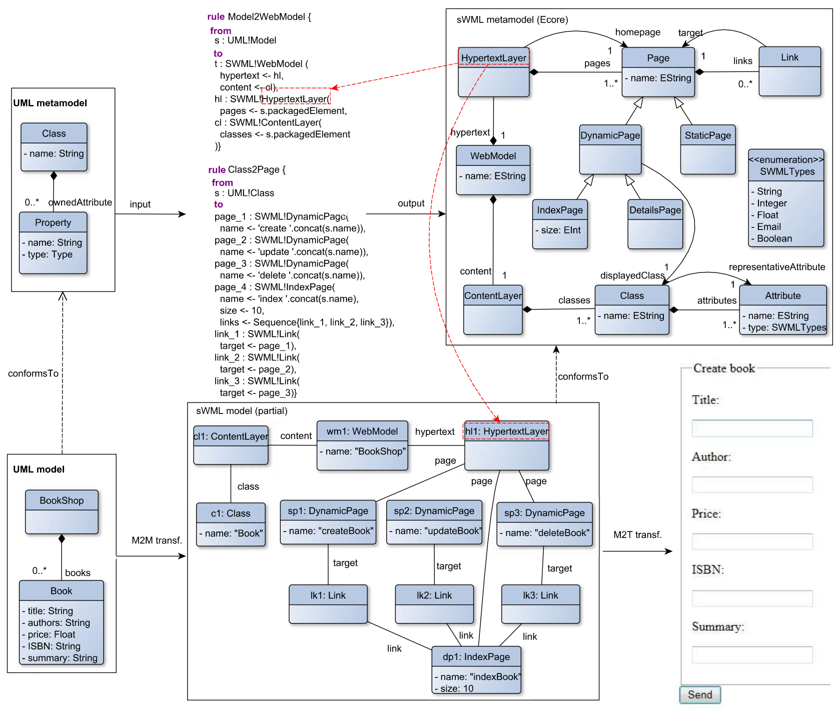
As an example of MDE infrastructure, we are going to use a very common *Forward Engineering* (FE) process, where models are used to design a solution that is later automatically transformed into a CRUD-based web application. This scenario is illustrated in Figure 2:

1. The left-bottom part shows a UML class diagram depicting the need to store information about books and bookshops.
2. A transformation uses this schema definition to generate a navigation model [6] with the usual CRUD pages as default website structure.
3. A final model-to-text transformation generates the code corresponding to the forms, pages and tables for the example.

The class diagram conforms to the UML language while the navigation model is represented as an object diagram conforming to a small DSL called sWML (*Simple Web Modeling Language* [5], inspired in IFML [6]). The transformation is written in ATL and describes how to generate sWML models from UML ones. The upper rule bootstraps the sWML model while the lower one iterates through the UML model and, for each class it founds, it creates the corresponding CRUD pages. More details on this example can be found in [1].

This transformation chain (from UML to sWML and from sWML to code) is implemented in the CD server (Jenkins) by creating two new jobs, one per transformation (see the last two jobs in Figure 3). This way we enable: the chaining of modeling tasks (M2T is automatically executed when M2M finalizes),

[4] One difference of running EMF standalone is that the application is unaware of plug-ins, so registrations have to be done now in the code



```
rule Model2WebModel {
 from
  s : UML!Model
 to
  t : SWML!WebModel (
   hypertext <- hl,
   content <- cl),
  hl : SWML!HypertextLayer(
   pages <- s.packagedElement,
  cl : SWML!ContentLayer(
   classes <- s.packagedElement
  )}

rule Class2Page {
 from
  s : UML!Class
 to
  page_1 : SWML!DynamicPage(
   name <- 'create '.concat(s.name)),
  page_2 : SWML!DynamicPage(
   name <- 'update '.concat(s.name)),
  page_3 : SWML!DynamicPage(
   name <- 'delete '.concat(s.name)),
  page_4 : SWML!IndexPage(
   name <- 'index '.concat(s.name),
   size <- 10,
   links <- Sequence{link_1, link_2, link_3}),
  link_1 : SWML!Link(
   target <- page_1),
  link_2 : SWML!Link(
   target <- page_2),
  link_3 : SWML!Link(
   target <- page_3)}
```

Fig. 2. Example of a model-based software process

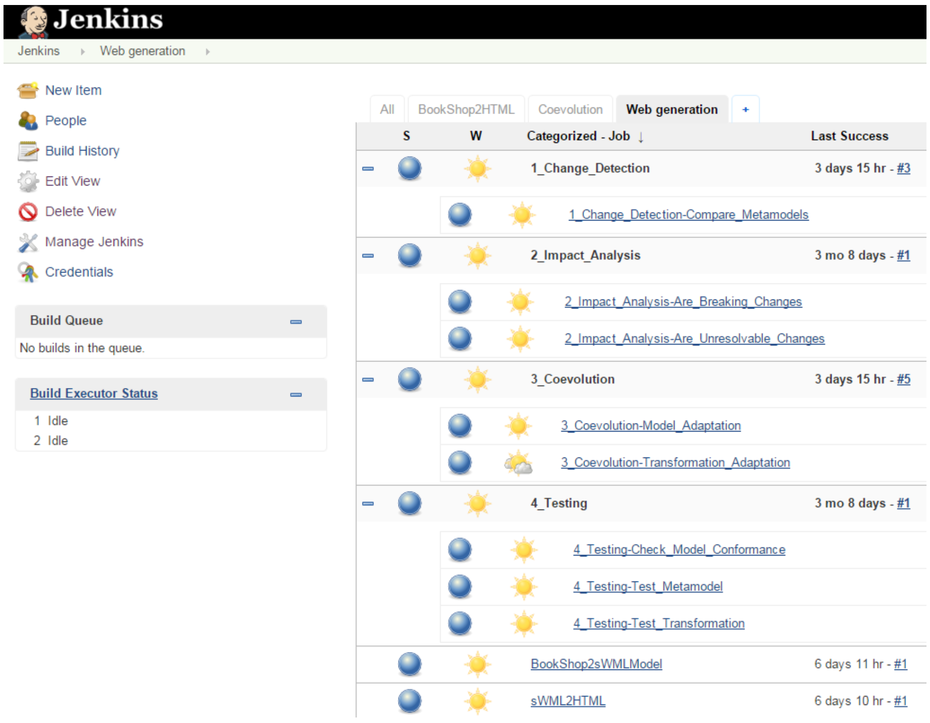
visualization of their status (if there is any error in the execution or resulting model or code) and the immediate re-execution of the process when a model is updated (either at the UML or sWML levels). Reactivity is achieved thanks to a hook between the SCM and Jenkins[5], that allows the execution of jobs as triggers after an update in the software repository.

Still, this integration is straightforward but quite dumb in the sense that the CD server sees models as pure text/XML artifacts and therefore is unable to use the model semantics to better manage the pipeline, for instance by preventing triggering the transformations when the model update does not have any real impact in the rest of the chain. We discuss a more advanced integration in the next section.

4 Continuous Evolution of MDE infrastructure

All MDE elements in the previous example are a software product on its own that have followed as well a build and deploy process, and therefore may benefit from being the target of a CD process themselves to bring all CD benefits to the

[5] https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin


Jenkins

Jenkins › Web generation ›

- New Item
- People
- Build History
- Edit View
- Delete View
- Manage Jenkins
- Credentials

Build Queue

No builds in the queue.

Build Executor Status

1 Idle

2 Idle

All | BookShop2HTML | Coevolution | **Web generation** | +

S	W	Categorized - Job ↓	Last Success
●	☀	1_Change_Detection	3 days 15 hr - #3
●	☀	1_Change_Detection-Compare_Metamodels	
●	☀	2_Impact_Analysis	3 mo 8 days - #1
●	☀	2_Impact_Analysis-Are_Breaking_Changes	
●	☀	2_Impact_Analysis-Are_Unresolvable_Changes	
●	☀	3_Coevolution	3 days 15 hr - #5
●	☀	3_Coevolution-Model_Adaptation	
●	⛅	3_Coevolution-Transformation_Adaptation	
●	☀	4_Testing	3 mo 8 days - #1
●	☀	4_Testing-Check_Model_Conformance	
●	☀	4_Testing-Test_Metamodel	
●	☀	4_Testing-Test_Transformation	
●	☀	BookShop2sWMLModel	6 days 11 hr - #1
●	☀	sWML2HTML	6 days 10 hr - #1

Fig. 3. Jenkins pipeline of the co-evolution process

MDE domain (or to best exploit them when part of a more global CD process). These benefits include:

- Reactivity: The co-evolution process does not have to be launched by the developer manually anymore. Everytime a new change is committed in the repository, the process will be triggered automatically.
- Parallelization: Different co-evolution solutions are given depending on the affected artifacts. Instead of applying them one by one, a CD server allows to execute in the same time all of them.
- Visibility: of the process and its state.
- Time saving: Co-evolution and testing is only executed if the impact analysis determines that it is actually needed.
- Flexibility: We may not need the modeling expert. The domain expert can execute the whole process alone.

Due to the complex and non-linear nature of MDE ecosystems, we must deal with changes at two different levels: the model level but also the metamodel one, not usually the case when developing more traditional software products where the grammars and libraries imported in the project hardly ever change during the development; instead in MDE, the DSLs change much more often. Covering this scenario is important to ensure the long-term maintainability of the MDE artifacts (and therefore of the software depending on them) as part of a *Continuous Evolution*[11] effort.

When co-evolving a MDE ecosystem, we must take into account the coupling between each single pair of artifacts. Dependencies between different artifacts in a MDE ecosystem can be seen in Figure 1. These are the most common coupling cases, that happen when the metamodel, which is the cornerstone of the ecosystem, evolves:

– Metamodel - model: When a metamodel evolves, instances of that metamodel have to be adapted to changes [8].
– Metamodel - transformation: Transformations are defined between metamodel elements, so when any of the metamodels of the transformation (source or target) evolves, it has to be adapted to that evolution as well [12].
– Metamodel - editors: When the metamodel defining the abstract systax of an editor changes, the rest of the editor artifacts are affected [9].

4.1 Evolution scenario: an example implementation

Coming back to our example of Figure 2, we propose a simple evolution scenario: the sWML metamodel evolves, renaming the name of the type *HypertextLayer* to *NavigationLayer*. This change forces us to change the references to that type in the transformations using it as input/output element and update all model elements that instantiate that type to reclassify them. We can see these impacts as dotted arrows in the figure.

In a naive MDE - CD integration (as the one sketched in the previous section), any change on a MDE artifact will trigger an update on all the depending elements which in turn could fire further changes down the lane. Ideally, the CD server should be smarter than that and be able to understand enough the MDE artifacts it manages in order to optimally coevolve them.

Figure 3 shows how our CRUD-based example has been implemented as a fully automated pipeline in a CD server (Jenkins). For clarity, jobs have been divided into phases:

1. **Change detection**: analyzing and classifying the kind of changes that have occurred after every update by comparing the two versions of the artifact. When the new version of the metamodel is committed, the process is triggered. Both metamodel versions will be compared by calling the tool EMFCompare in charge of generating a difference model that represents the differences between the two versions of the metamodel. In this case, it will result in a *Rename Class* type of change. Notice that in this step, a textual comparison tool is not enough: a tool that deals with model semantics is needed.
2. **Impact analysis**: it is assessed what parts of the system are likely to be affected by a change on the related artifacts running an impact analysis algorithm. It decides, for each depending artifact, whether the changes should be classified into:

   - *Non Breaking Changes (NBC)*: changes that do not have any impact.
   - *Breaking and Resolvable Changes (BRC)* changes that have an impact but that can be resolved automatically.
   - and *Breaking and Unresolvable Changes (BUC)* changes that have an impact that requires human intervention.

   As we can see in Figure 3, there is one job for detecting breaking changes and another for unresolvable changes. The impact analysis has been implemented as an ATL transformation wrapped in Java. The *Rename Class* would be classified as a BRC type.
3. **Synchronization**: Once we know the affected artifacts (and the kind of changes relevant to them), they are synchronized: for NBCs, the CD server should not propagate anything, for BRCs it should evolve the depending element automatically and for BUCs mark it as in an erroneous state for manual reviewing. This step is implemented via an ATL transformation wrapped in Java. In the example, as it is a BRC, the co-evolution jobs will adapt both the models conforming to the *sWML* metamodel and the model to model transformation. In the case of models, *HypertextLayer* elements will be renamed; and in the ATL transformation, elements of type *OclModelElement* will be renamed as well. This is because there is a coupling between *EClass* element in Ecore and *OclModelElement* in ATL metamodel. As we can see in the pipeline, there is one job for coevolving models and another one for coevolving transformations.
4. **Testing** the results. Conformance verifications have been implemented using EMF default checking mechanisms. In the pipeline, there are several testing jobs, one per artifacts: model, metamodel and transformation.

The corresponding jobs have been linked using the *post-build* section mechanism provided by Jenkins, where all elements are tested after any change and feedback is provided if any error is detected (see [1] for the full details).

As we can see, this smarter integration would save a lot of time and, potentially, many unnecessary redeployments in any non-trivial system. In the case of Forward Engineering scenarios where code is generated, we avoid all the generation and testing of code.

Nevertheless, from the naive to the smart integration approaches we have a full range of intermediate solutions depending on the characteristics of the project and the availability of the model-based components required for each of the four previous tasks in the specific project context.

4.2 Adoption levels

Therefore, a step-wise adoption of Continuous MDE for software companies could follow the phases described next, which progressively raise the level of adoption:

Using generic support Without any specific model support, the CD server treats models as plain text and is not aware of their structure at all. Dependencies

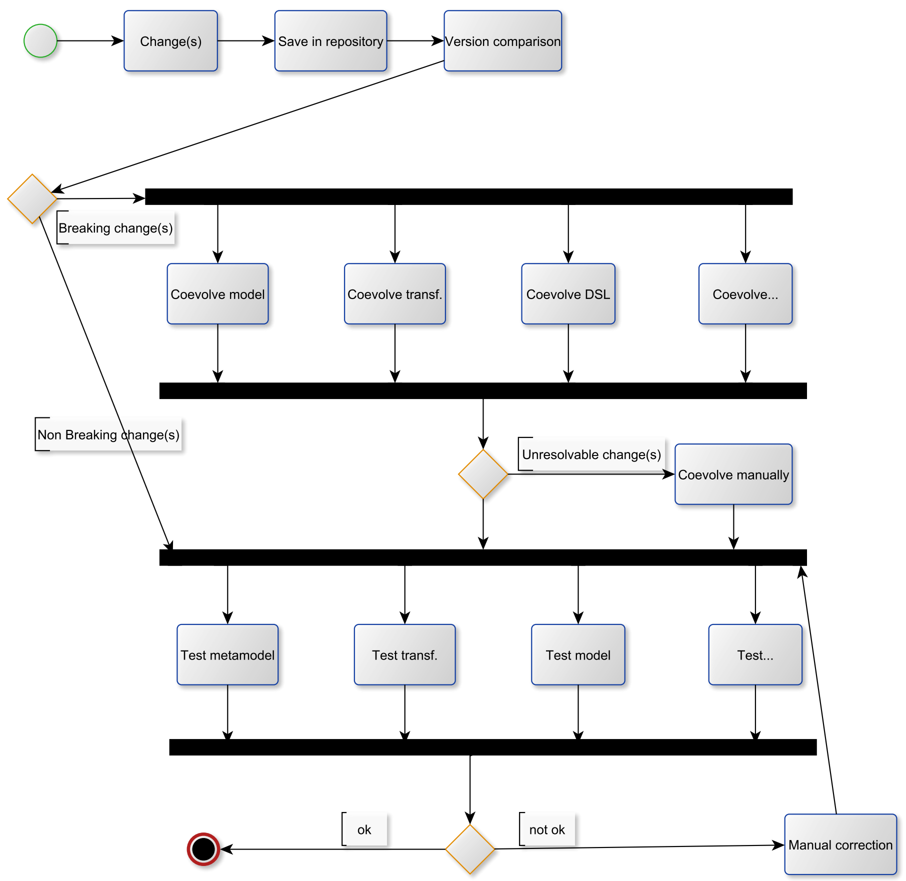
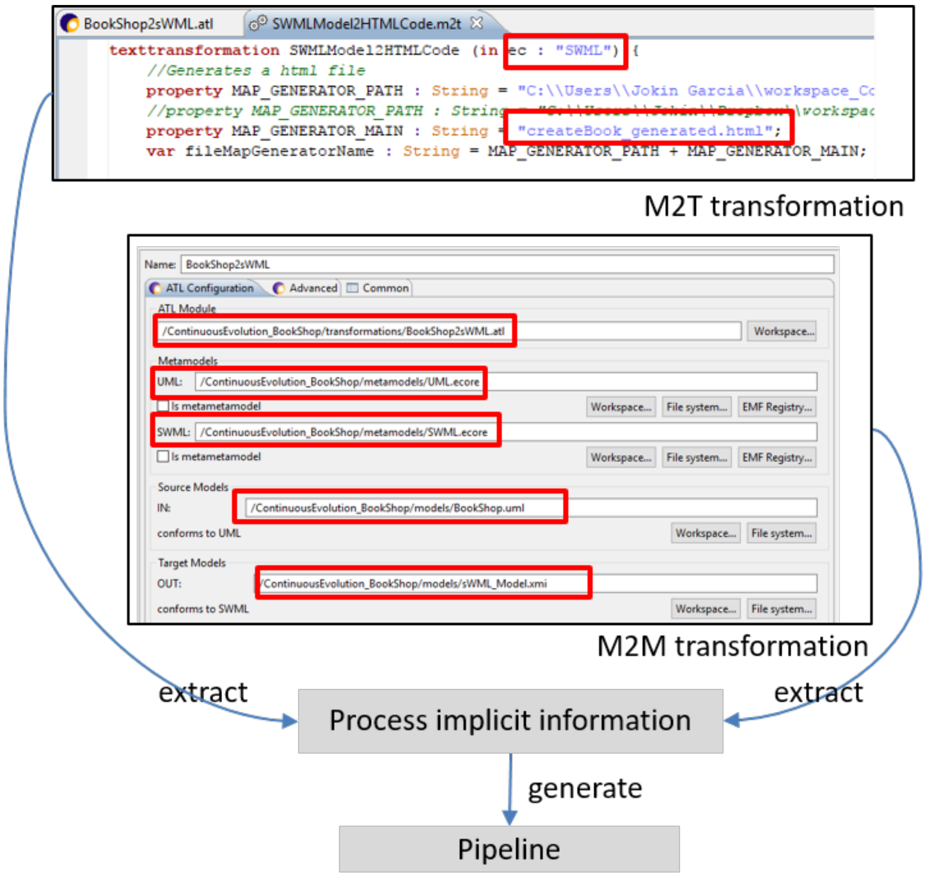


Fig. 4. Conditional and parallel execution of coevolution jobs

between jobs have to be manually added and co-evolution is limited to alerting developers when an element needs to be manually reviewed. Any new model version triggers all depending jobs.

With co-evolution support We can add co-evolution support for coupled MDE artifacts. As described in the evolution scenario and (see also the generic process described the Figure 4), a model comparison job is triggered to interpret model changes for a given artifact when a new version of the model is saved in the repository.

– If there are breaking changes, co-evolution jobs (one for each coupled element type) take care of processing those changes and determining whether the depending elements need to be resynchronised. Those jobs are parallelizable.



```
texttransformation SWMLModel2HTMLCode (in ec : "SWML") {
    //Generates a html file
    property MAP_GENERATOR_PATH : String = "C:\\Users\\Jokin Garcia\\workspace_Cc
    //property MAP_GENERATOR_PATH : String = "C:\\Users\\Jokin\\[illegible]\\workspac
    property MAP_GENERATOR_MAIN : String = "createBook_generated.html";
    var fileMapGeneratorName : String = MAP_GENERATOR_PATH + MAP_GENERATOR_MAIN;
```

Name: BookShop2sWML

ATL Module: /ContinuousEvolution_BookShop/transformations/BookShop2sWML.atl

Metamodels
UML: /ContinuousEvolution_BookShop/metamodels/UML.ecore
Is metametamodel ☐
SWML: /ContinuousEvolution_BookShop/metamodels/SWML.ecore
Is metametamodel ☐

Source Models
IN: /ContinuousEvolution_BookShop/models/BookShop.uml
conforms to UML

Target Models
OUT: /ContinuousEvolution_BookShop/models/sWML_Model.xmi
conforms to SWML

Fig. 5. Extraction of transformation execution configurations

Then, if there was any unresolvable change, developers will be notified to do a manual co-evolution. After that, tests will be executed in parallel.

– If there are not breaking changes, tests are executed directly, without passing through the co-evolution step.

This conditional and parallel execution shortens the deployment time. The limitation of this approach is that jobs are still manually added as part of the pipeline definition phase but the pipeline is automatically executed (except for BUCs) afterwards.

Automatizing the process As a final step in the integration between MDE and Continuous Delivery, we could automate the definition of the pipeline itself. The initial configuration of the CD server can be generated programmatically.

For instance, the pipeline feature in Jenkins 2.0 (or Job DSL plugin[6] in previous versions of Jenkins) allows to build Jenkins jobs using a simple DSL on top of Groovy, which can be integrated in version control systems. This script, in turn, could be generated based on the analysis of the (implicit or explicit) dependencies between artifacts like a transformations configuration launch with specific reference to the input/output metamodels. In Figure 5 we can see the implicit process information in the m2t and m2m transformations regarding the relation between transformation, metamodels and models. We could take advantage of this information for the impact analysis and co-evolution phases, where it is needed to know the coupling between artifacts. In this scenario, both the definition and execution of the pipeline are fully automated. Implementing this part is left as future work.

5 Related Work

Co-evolution of artifacts in the MDE ecosystem has been tackled in several works, where specific solutions have been proposed depending on the type of artifact to be co-evolved. It has been studied the impact of metamodel evolution on models [8], transformations [12] and editors [9]. But, as far as we know, there are not works with an holistic and automatized view of the evolution. They are limited to an on-demand and manual co-evolution between pairs of artifacts. Using those evolution tools as building blocks, we are proposing a more ambitious approach where the co-evolution is reactive, automatic and parallelizable. Moreover, the process can be implemented with existing tools, integrating all the modeling tasks that are standalone.

The most basic premise in order to apply any kind of Continuous* practice is that all the artifact versions are committed to a Version Control System. VCSs ([7], [16], [14]), comparison ([21]) and merge tools ([17]) for models have been proposed.

There are also methodologies based on *Ant* to chain MDE operations [18] but they have not been proposed as part of a CD process.

Papers studying the synergies between CD and MDE for specific domains have also been presented. In [3], authors provide a model-based approach to generate TOSCA blueprints (that supports the definition of deployments as code), allowing the quick (re) deployments of cloud applications. Also in the domain of cloud computing, [20] proposes a model-driven approach to abstract and automate a continuous delivery process of cloud resources. This is done with a tool that uses a Domain Specific Language (DSL) to model the cloud infrastructure and a transformation that from that model creates scripts to manage different Configuration Management Tools. Similarly, in [10], a developer team can specify a model of the deployment of its application and automatically enact it in a test environment. Finally, in [13], authors present a prototype that uses a model-driven generator combined with CI server. They report on an empiri-

[6] https://wiki.jenkins-ci.org/display/JENKINS/Job+DSL+Plugin

cal evaluation that shows the benefits of using MDE in combination with a CI server.

Complementary to these approaches, our work provides a more generic solution and studies the benefits of using CD processes and tools in the maintainability of MDE infrastructures themselves.

6 Conclusions and Future Research Directions

We have sketched the integration of MDE artifacts as first-class citizens in continuous software engineering, ranging from a direct use of current integration platforms to advanced coevolution scenarios, depending on the needs of the project and the role MDE plays in it. This benefits developers of both software artifacts (that can benefit from MDE) and MDE artifacts (that can benefit from CSE in their work).

Nevertheless, to achieve a complete and smooth support for MDE in CSE, we need to extend the state of the art in several directions. First, MDE technologies themselves need to become more mature. While some (e.g. model transformations) are reliable and ready-to-use in complex industrial scenarios, others (e.g. model merging) require more work to provide automatic solutions and/or professional tools. Secondly, CI components should be model-aware, providing default support for some model management operations (like model comparison for well-known types of models, e.g. UML class diagrams) or at least standard extension points to provide that. Finally, brand new research proposals should target some of the co-evolution scenarios and smarter dependency and impact analysis algorithms that have not been addressed so far and that would enable a better CSE automation for MDE projects.

We hope to see progress in these directions in the coming years.

References

1. https://github.com/jokingarcia/ContinuousEvolution. access 9.07.2018.
2. Scott W. Ambler. Agile software development. In *Encyclopedia of Software Engineering*, pages 29–46. Taylor & Francis, 2010.
3. Matej Artač, Tadej Borovšak, Elisabetta Di Nitto, Michele Guerriero, and Damian A. Tamburri. Model-Driven Continuous Deployment for Quality DevOps. In *Proceedings of the 2Nd International Workshop on Quality-Aware DevOps*, 2016.
4. Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile Software Development, 2001.
5. Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice.* Morgan & Claypool Publishers, 2012.
6. Marco Brambilla and Piero Fraternali. *Interaction Flow Modeling Language.* Morgan Kaufmann, 2015.
7. Petra Brosch, Gerti Kappel, Martina Seidl, Konrad Wieland, Manuel Wimmer, Horst Kargl, and Philip Langer. Adaptable Model Versioning in Action. In *Modellierung*, 2010.

8. Andreas Demuth, Markus Riedl-Ehrenleitner, Roberto E. Lopez-Herrejon, and Alexander Egyed. Co-evolution of Metamodels and Models through Consistent Change Propagation. *Journal of Systems and Software*, 111:281–297, 2016.
9. Davide Di Ruscio, Ralph Lämmel, and Alfonso Pierantonio. Automated Co-evolution of GMF Editor Models. In *Software Language Engineering*, 2011.
10. Nicolas Ferry and Arnor Solberg. *Models@Runtime for Continuous Design and Deployment*, pages 81–94. Springer International Publishing, 2017.
11. Brian Fitzgerald and Klaas-Jan Stol. Continuous Software Engineering: A Roadmap and Agenda. *Journal of Systems and Software*, pages 1–14, 2015.
12. Jokin García, Oscar Diaz, and Maider Azanza. Model Transformation Co-evolution: A Semi-automatic Approach. In *Software Language Engineering*. Springer, 2013.
13. Vicente García-Díaz, Jordn Pascual Espada, Edward Rolando Núñez-Valdéz, B. Cristina Pelayo García-Bustelo, and Juan Manuel Cueva Lovelle. Combining the Continuous Integration Practice and the Model-Driven Engineering Approach. *Computing and Informatics*, 35:299–337, 2016.
14. Taid Holmes, Uwe Zdun, and Schahram Dustdar. MORSE: A Model-Aware Service Environment. In *4th IEEE Asia-Pacific Services Computing Conference*, 2009.
15. Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Pearson Education, 2010.
16. Maximilian Koegel and Jonas Helming. EMFStore: a Model Repository for EMF Models. In *International Conference on Software Engineering*, 2010.
17. Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. *Merging Models with the Epsilon Merging Language (EML)*, pages 215–229. Springer Berlin Heidelberg, 2006.
18. Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. A Framework for Composing Modular and Interoperable Model Management Tasks. In *MDTPI Workshop*, 2008.
19. Richard F. Paige, Nicholas Matragkas, and Louis M. Rose. Evolving Models in Model-Driven Engineering: State-of-the-art and Future Challenges. *Journal of Systems and Software*, 111:272 – 280, 2016.
20. Julio Sandobalin, Emilio Insfrán, and Silvia Abrahão. An Infrastructure Modelling Tool for Cloud Provisioning. In *International Conference on Services Computing*, pages 354–361, 2017.
21. Antoine Toulmé. Presentation of EMF Compare Utility. In *Eclipse Modeling Symposium*, 2006.
22. Jon Whittle, John Hutchinson, and Mark Rouncefield. The State of Practice in Model-Driven Engineering. *IEEE Software*, 31:79–85, 2014.
23. Jon Whittle, John Hutchinson, Mark Rouncefield, Hakan Burden, and Rogardt Heldal. Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem? In *International Conference on Model Driven Engineering Languages and Systems*, 2013.