Performing Incremental Generation of Code in Model-driven Engineering

Vicente García-Díaz, B. Cristina Pelayo G-Bustelo, Oscar Sanjuán-Martínez and Juan Manuel Cueva Lovelle
University of Oviedo, Department of Computer Science, Sciences Building, C/Calvo Sotelo s/n, 33007 Oviedo, Asturias, Spain
Email: {garciavicente, crispelayo, osanjuan, cueva} @uniovi.es

The software development approach called Model-Driven Engineering continues to evolve at a rapid pace. A key aspect is the automatic generation of artifacts at lower levels of abstraction. However, this process typically does not take into account the evolution of systems throughout their life cycle; moreover, it is done in an unwieldy and repetitive manner. There are some works that address the problem of incremental generation of artifacts, but unfortunately that research tends to be focused on generating artifacts in the form of a model, instead of generating source code of an application that may already be deployed and running. Other studies address the issue in a limited way. In this paper, we present a proposal, a prototype, and a case study for incrementally generating source code artifacts from models. This work aims to minimize the impact of changes on applications that may already be deployed.

Keywords: Model-driven engineering; evolution; incremental generation; traceability

ACM Classification: D.2, D.2.11, D.2.13

1. Introduction

Model-Driven Engineering (MDE) (Kent, 2002) is a relatively new paradigm in software development that has gained importance since its emergence, especially in the development of software product families (Clements and Northrop, 2002). It consists of using models (Seidewitz, 2003) as key elements in development. Therefore, it achieves a higher level of abstraction than that achieved by traditional programming languages (e.g., C++ and Java). In general, these models are automatically transformed, reducing the level of abstraction to artifacts such as Java source code or documentation in HTML format, thus facilitating the generation of software systems (Völter and Stahl, 2006). Very often, models are also transformed into other models of a lower level of abstraction before being transformed into code as in the case of the Model-Driven Architecture (MDA) (Miller et al., 2003) standard, in which a platform independent model is transformed into platform specific models, and finally into implementation specific models. MDE then, proceeds as follows: a developer creates a model and a transformation engine or a generator is responsible for creating the artifacts that correspond to the model. These artifacts could be, for instance, other models representing any software view, a software product, or any software component.

Models are therefore subject to the same challenges that any software system may suffer during its evolution (Mens et al., 2005); they are also sensitive to changes throughout their life cycles,
Performing Incremental Generation of Code in Model-driven Engineering

which may result, for example, from shifts in the customer business. Typically, if the developer modifies the model from which the artifacts are generated, he/she should run the generator to regenerate all artifacts, including those that were not developed anew for the specific model under study. From the point of view of code artifacts, this is a problem in several ways. First, it is difficult to maintain applications that have already been deployed. Second, it is impossible to make the generated artifacts traceable. Third, it generally does not optimize the computing resources that are available.

The incremental generation of artifacts in MDE refers to the ability of a generator to avoid rebuilding all artifacts every time a change is made in a model, using models as the key elements from which software systems are built.

Figure 1 shows a sequence of 3 steps to illustrate how an incremental generator of artifacts works. Those steps, from 1 to 3, represent a possible evolution of a system model that generates source code artifacts. For simplicity, we assume that each diagram element leads to the generation of a Class of the same name in an object-oriented programming language such as Java. In step 1 (i.e., the initial case) four Classes (i.e., A, B, C, and D) are generated from the upper left diagram. However, in step 2, Classes C and D are deleted because the elements that represent them no longer appear in the diagram. In step 3, a hypothetical Class E is added, because an element to represent this Class has been included in the diagram along with a change made in Class B, represented by B'.

We can see that the scheme shown in Figure 1 is very likely to save time in the generation of artifacts as well as to facilitate changes in applications already deployed, thereby reducing the

Figure 1: Sample Sequence of the Incremental Generation of Artifacts
negative impact of changes in a model. Otherwise, the generator would always have to delete and regenerate all artifacts with each change in the source model. Moreover, by maintaining different versions of the model, traceability can be achieved so that it is possible to know which and when artifacts have been added, deleted, or changed. As such, this is a powerful tool for maintenance and debugging.

In summary, with incremental generation of artifacts in MDE:

• With the initial model, the four classes represented in the upper right of Figure 1 have been generated. Obviously, this first step is exactly the same, whether or not there is incremental generation of artifacts.

• After the first change in the initial model, two Java classes have been left intact and the other two classes have been deleted. Without incremental generation, the four classes would have been deleted before recreating them.

• After the second change, one class has been added and another class has been changed. Without incremental generation, the two classes that existed after the previous step would have been deleted before recreating from scratch the three final classes based on the information provided by the model.

Thus, without incremental generation of artifacts:

• If anyone changes a model that serves to create artifacts of a software system that is already deployed, it is absolutely necessary to stop the application, generate all the artifacts back, and finally redeploy the application. With incremental generation of artifacts, depending on each case, the impact on a deployed system could be much lower.

• Although all versions through which a model passes could be saved, nobody could ever trace neither the changes that have occurred between different application versions nor when they have been produced, since it is always necessary to regenerate all the artifacts.

• As always it is necessary to delete everything and rebuild all the artifacts, the computational resources required are inevitably higher than with the incremental generation of artifacts.

Despite its benefits, the incremental generation of artifacts is an issue that is still little addressed in the scientific literature. Thus, the objective of this paper is to offer a proposal to achieve a successful generation of source code artifacts using an incremental approach. In order to create a generator with this feature and a study case, we have developed a tool based on standard technologies. The remainder of this paper is structured as follows: In Section 2, we present the state-of-the-art of incremental generation of artifacts using the MDE paradigm. In Section 3, we propose an incremental MDE solution and we describe the main features of the developed prototype. Section 4 shows the obtained results based on a case study, and finally, we discuss the results and future work to be done in Sections 5 and 6.

2. State-of-the-Art

Incremental generation of artifacts requires knowing the changes that have occurred in a model with respect to another version of the same model, that is, it is necessary to know the differences among the states through which a model passes. This knowledge has a lot of distinct applications. For instance, there are a considerable number of studies focusing on the need to develop Model-specific Version Control Systems (MVCS) (Oliveira et al., 2005). Works such as Reiter et al. (2007) and Altmanninger et al. (2008) have studied optimistic source management in which it is essential
Performing Incremental Generation of Code in Model-driven Engineering

that matching algorithms are used to represent differences and/or perform unions and merges between models (Bendix and Emanuelsson, 2008). Other works such as Lin et al (2004) have demonstrated the importance of the representation of differences between models as a way to check the correctness of the specification of transformations between models made by humans. Thus, if there are no differences between the model expected after treatment and that actually obtained, the transformation is considered well defined. Garcés et al (2009) used the difference between various versions of a metamodel in order to adapt the models to conform to a new metamodel version. Other studies have also cited scenarios in which these techniques are useful (Selonen, 2007). These scenarios include the analysis of the evolution of software and the detection of model inconsistencies.

Knowing the difference between models is not a simple process (Read and Cornell, 1977) and comprises three phases (Brun and Pierantonio, 2008). The first two stages (i.e., the calculation of differences between models and the representation of these differences) are particularly useful, but the third stage, which involves visualizing differences in a format that the user can easily read, is not strictly necessary for the purposes of this work because the artifacts are automatically generated.

Therefore, to generate artifacts in an incremental way, we believe that there are several required activities represented in the Figure 2:

- Calculate the difference between two versions of a model that conforms to the same metamodel (sometimes known as delta).
- Correctly represent the difference.
- Generate artifacts strictly on the basis of the difference represented.

Currently, there are many model-driven tools to generate artifacts. However, in terms of incremental generation, deficiencies were found in the tools that we reviewed, including those that have been widely cited in works such as Palacios et al (2008). Several tools offer incremental generation features, they do so in a limited way.

For example, EclipseUML (http://www.uml2.org/) is a proprietary tool built into Eclipse, which facilitates the design of software systems using various types of UML (OMG, 2007) diagrams. Using some of the elements of the diagrams, the tool generates Java source code that remains synchronized with changes either in the code view or in the diagram view. In addition, OptimalJ (http://www.compuware.com/) uses a synchronization mechanism called active synchronization. When a developer changes a UML model, the model and the previously-generated J2EE artifacts are synchronized. Another example is objectiF (http://www.microtool.de/objectiF/), which

![Figure 2: Key Aspects in Incremental Generation of Artifacts](image-url)
regenerates only the components that are changed in a given model. All of them have shortcomings, as they are very technology-specific, they only allow changes in design time and they do not take into account that a change in one part of the model may affect other parts.

The following paragraphs will show the state-of-the-art of current technologies that will help understand why none of the current generators can create source code in an appropriate manner.

2.1 Incremental Development Approaches

According to Hearnden et al (2006), there are two approaches to model-driven incremental software development, which are the re-transformation and the live model transformation.

With the re-transformation (upper left of Figure 3), each increment or delta (Δ) in the source model requires performing a new complete transformation, leading to a new target model. Thus, there is not an incremental update in the target model, but only in the source model. Moreover, to evolve a model based on another, we would have to perform a join between the two latest versions of the created artifacts, which can be extremely complex. This strategy is very basic as it...
rebuilds all the artifacts from the model, which is often not taken into account as a true incremental approach.

The live model transformation (upper right of Figure 3), for its part, saves the execution context of the first transformation performed on a model and continues using it in the following transformations, which implies that it is a ongoing process that does not have a specific end. So, there is no need to perform the complex join between the target models because partial transformations are being performed only in accordance with the variations in the source model. The delta of the source model is transformed into a delta for the target model, which causes that only the strictly required elements of the target models are modified. There are some proposals to perform live model transformations. For instance, the proposal presented in Hearnden et al (2006) uses a declarative transformation language based on rules and is tightly integrated with the internal structure of the transformation engine called Tefkat (Lawley and Steel, 2006). The idea is to extend the transformation engine based on the standard mechanism for the interpretation of logical languages, that is, the Selection, Linear, Definitive (SLD), which is a restriction of the general resolution principle (Robinson, 1965). Ráth et al (2008) shows other work belonging to the category of live model transformation. The novelty of such a work is that it presents an incremental approach based on the graph pattern matching and managing complex transactions. To that end, a model transformation framework called VIATRA2 (Varró and Balogh, 2007) is employed, which uses the RETE network algorithm (Forgy, 1990). For the user, the language is based on the definition of restrictions, conditions, rules to define elementary manipulations, and rules to describe control structures. The main disadvantage of this approach is that it needs to always keep available the execution context, which can be computationally expensive and is more prone to inconsistencies, especially in networked systems. In addition, other works (e.g., van der Meij, 2009) have appeared which also take into account other types of approaches, the coarse-grained transformation and the fine-grained transformation:

The fine-grained transformation (lower left of Figure 3), which consists of performing transformations from the delta, only when the user wants, avoiding the need of saving the execution context at any time. This approach is more interesting than the live model transformation because it does not require adapting the current generation engines, but it is limited because it can only generate artifacts that are directly related to the delta. That is, it cannot generate any artifact whose information is not contained in the delta.

Finally, the coarse-grained transformation (lower right of Figure 3), which can be seen as an extension of the previous approach. A drawback is that it is also necessary to use parts of the source model, even when they have not been modified as a result of the model evolution. The aim is to regenerate artifacts for which more information is needed than that available in the delta, which is a key aspect when there are side effects on changes in the source model. This approach has two phases:

- It uses the fine-grained transformation up to a certain depth.
- From the previous level of depth, it uses the parts of the source model that have not been modified.

These steps make the approach rigid to the developer and that not all the information is available, which in some cases may be necessary.
2.2 Obtaining the Delta

There are a number of algorithms that could be used to compare models, and thus obtain the delta between them. The first algorithms were developed to compare only text files (e.g., Hunt and McIlroy, 1977; Hirschberg, 1977; Myers, 1986). Although these proposals serve their purpose, they use a unit of version based on a file and a unit of control based on the paragraph level, incorrect abstractions for working with models because they use different data structures (Oliveira et al., 2005).

From the point of view of models, the simplest approach is to assume that each element has a persistent Universally Unique Identifier (UUID) assigned at the time of its creation (e.g., Alanen and Porres, 2003; Farail et al., 2006). This assumption is very optimistic because it cannot be applied to models that are built independently of each other or in tools that do not support it. In addition, the use of UUIDs does increase the size of the models. However, in cases in which it can be applied, it allows comparison of models quickly and easily.

The next step was the use of a technique based on dynamic identifiers calculated on the basis of certain values of the model by using functions defined for that purpose (Reddy and France, 2005). This approach allows us to compare models that have been constructed independently but it has the disadvantage that it is not easy to define functions for comparing each model element for each specific problem that may exist.

There are also approaches that use trees with ordered children (e.g., Tai, 1979; Cobena et al., 2001) or disordered children (e.g., Zhang and Shasha, 1989; Wang, 2005). Trees are a special type of graph that cannot contain cycles.

However, the nature of models makes the trees too restrictive to represent them. Thus, the latest generation of algorithms is based on treating the models as graphs, which is not a trivial problem (Khuller and Raghavachari, 1996), and to make comparisons by looking for similarities between elements of models using heuristics. To do this, there are mainly two groups of algorithms: independents of the metamodel and dependents of the metamodel. The latter can deliver better results because they are much more specific focusing on a particular problem.

Comparison Depending on the Metamodel

Nejati et al. (2007) proposed a series of algorithms for matching using heuristics in a terminological, structural, and semantic level. They worked only with state diagrams that described behaviour issues for the telecommunications industry. Ohst et al. (2003) did a work that was performed to obtain the differences between versions of UML diagrams making strong emphasis on the representation of these differences.

The main disadvantage of these works is that the proposed algorithms and tools, in which they are implemented, are specific to the type of model. The same drawback is found in other works such as Xing and Strouilia (2005) or Selonen (2007). The fact of focusing on UML or any other specific metamodel as the Ontology Definition Metamodel (ODM) (OMG, 2005c) makes the algorithms restricted only to the chosen metamodel. Moreover, models from an arbitrary Domain- Specific Language (DSL) (van Deursen et al., 2000) are typically more general than, for example, UML models. Thus, they often have other structure, syntax, and semantics (Lin et al., 2007), requiring a different treatment.

Comparison not depending on the metamodel

There are several works whose purpose is to show how to compare models following a meta-
Performing Incremental Generation of Code in Model-driven Engineering

model independent approach. For example, Lin et al (2007) presented an algorithm and a tool (DSMDiff) that calculates the differences and similarities between two models of an arbitrary DSL and is based on the Generic Modeling Environment (GME) (Ledeczi et al, 2001). Other interesting work in this field was done by Melnik et al (2002), who presented a matching algorithm based on a fix point computation that is usable across different scenarios; Mandelin et al (2006), who presented a framework based on Bayesian methods for finding model correspondences automatically; Selonen and Kettunen (2007), who presented a flexible approach for inferring correspondences between model elements; Treude et al (2007), who presented a technique for computing differences between models an order of magnitude faster than other algorithms using a high-dimensional search tree for efficiently finding similar model elements; or Rivera and Vallecillo (2008), who presented a metamodel to represent differences between models using also a model as a central element.

2.3 Representing the Delta

To represent the differences between models, various techniques have emerged in recent years. Cicchetti et al (2007b) have listed a number of features that any technique for representing differences between models should have. These include metamodel independence and model-based design. Thus, a representation should not be based on a specific metamodel; instead, the use of metamodels should be tailored to each problem. In fact, a commonly accepted solution is the use of a four-layer architecture in which the solution with the higher level of abstraction is a single root meta-metamodel (e.g., the Meta-Object Facility (MOF) (OMG, 2005a) Modeling Space (Djuric, 2006)). Frankel (2003) has also pointed out that the use of a common root metamodel is critical for the success of MDE. With respect to model-based design, a commonly-accepted principle is Everything is a model by Bézivin (2005). This principle, together with the use of metamodels and meta-metamodels, makes available numerous tools under the MDE paradigm through a higher level of abstraction.

Edit Scripts

One of the techniques used to represent the differences between models is edit scripts, which are presented in works such as Alanen and Porres (2003). Edit scripts create sequences of primitive operations that describe the modifications that a model has suffered and are based on a sequence of transformations that can represent the difference as a delta (Δ). To describe the delta, edit scripts use elementary transformations that must comply with certain restrictions that limit the use of this technique. These transformations, which describe the changes of a model with respect to another model, include new, del, set, insert, remove, insertAt, and removeAt.

Colouring

Another commonly used technique is colouring. This technique is discussed in works such as Ohst et al (2003) and consists in graphically showing the differences by using a diagram. This diagram contains the common parts between the source and the target models and marks the parts that have changed. Another example is the EMF Compare interface (Toulmé, 2007).

The Difference Model

With respect to the techniques used to represent the difference between models, the last technique is discussed in the work of Cicchetti et al (2007b). Given two models that conform to the same
metamodel, the difference between them can be used to generate another model that conforms to a metamodel derived from the first one. Thus, the model arising from the differences will be a first class artifact in the build process and will capture the differences of the second model compared to the first one, because elements may have been changed, deleted or added (Figure 4). Through the use of this technique, the difference between models is itself a model that conforms to a metamodel, which in turn can work with any tool or MDE technology.

3. Proposed Solution

Our proposal provides the following features, that none of existing tools can fully support:

• Generation of artifacts according to the evolution. Current tools generate artifacts according to the input in a predetermined way. As such, they do not take into account whether an item in the model has been changed, added, deleted, or whether it was already in the latest version of the model.

• Separation between modeling and generation. In many existing tools, model and artifact views are combined in a development environment. However, users do not necessarily want or need to generate artifacts when they model their business. For example, continuous integration tools (Herbsleb and Grinter, 1999) make a clear distinction between sources (i.e., models) and the generation of artifacts.

• Generation independent of platform. Many existing tools only work with a specific architecture and platform (e.g., J2EE).

• Flexible generation of artifacts. Some existing tools generate artifacts in a rigid manner, making it impossible to change the artifacts that are generated from the input model. This makes generation too rigid to adapt to new requirements.

• Flexible metamodels. It is not flexible enough to be restricted to a single metamodel (e.g., UML) without the possibility of incorporating other metamodels.

• Free and open implementation. Several tools use proprietary technologies that preclude the study of their internal architecture.
3.1 Development Approach

Figure 5 shows the approach proposed in this work to solve the problems of the current methods for generating artifacts incrementally. Such a proposal aims to avoid the problems that we have found in the up-to-date approaches. The basic idea is to have in a single place the information of the delta of two consecutive states of the model, together with the information of the intersection between both of them; that is, the information that has not changed. In other words, it contains in a single element, the Unified Difference Model (UDM), the same information as the source model after the changes that have occurred but appropriately classified to facilitate the incremental generation of artifacts. Later, we will explain why this separation is useful.

Table 1 shows a comparison between the different existing approaches and the proposal made in this work to generate artifacts incrementally according to various aspects that may be important from the point of view of the transformation:

1. Impact on previously generated models. The re-transformation forces to completely regenerate the target model, which makes a great impact on a previously generated model. This reason does not make it possible to truly perform incremental developments.

2. Efficiency. The re-transformation leads to an unnecessary waste of resources, as it completely regenerates the target model for all occasions. To small changes in large models, that are the most common when the system is stabilized (van der Meij, 2009), such an action can waste a great deal of time and computational resources.

3. Traceability. The re-transformation does not take into account the incremental growth of the target model, thus precluding the implementation of the traceability of the system evolution. Then, it is not possible to know who, where, and what model elements have been added, removed, or changed during the lifecycle of the models. Traceability is a powerful tool for managing the evolution of any software system (Gotel and Finkelstein, 1994).

4. Execution context. The live model transformation is based on the use of a structure that saves the exact state of the evolution of a model. Thus, it forces to always have a link, maintained by the transformation engine, between the execution context and the models. In real cases in which the models are very large, that can lead to high consumption of resources even though...
it is possible that no transformation is going to be performed for a long time. In addition, it is common the use of distributed technologies where a failure can lead to the loss of information about the model evolution, making the structure unusable for any further requests. The use of an execution context is interesting but it is necessary to solve many problems before a possible implementation in a large scale.

5. Transformation engines. To save the execution context, proposals of live model transformations create ad-hoc execution engines or perform major modifications to existing ones (e.g., Ráth et al, 2008). This requires extra work, avoids using many existing tools, and often leads to solutions far from industry standards.

6. Independence. The coarse-grained transformation does not contain all the semantic and structural information in a single logical and physical component, but it is contained in two. At first glance it could not seem a problem, but this leads to problems for performing transformations or any other operation on the delta and the source model with the currently available tools. Furthermore, it is also much less intuitive, concise and compact (Konemann, 2009).

7. Transformation of non-involved elements. The fine-grained transformation has the disadvantage that if a change in the source model causes the regeneration of part of the target model from an element of the source model that has not changed, such a task cannot be performed because we only have the information of the delta.

8. Information availability. The fine-grained transformation only has the information of the delta, which makes it an approach too limited for some complex transformations. For its part, the coarse-grained transformation could provide all the necessary information, but it would depend on the tools used and on their configuration, but not on the transformations as would be most desirable.

All the above suggests that the solution with greatest advantages is the direct transformation that we used in our case study.

3.2 Obtaining the UDM

Given the importance of the reuse of knowledge and experience, our proposal is based on techniques already being used and investigated in other areas of MDE. Incrementally generating artifacts at the very least requires knowing what modifications have occurred in a version of a model with respect to a previous version; that is, it requires knowing the different stages of the evolution of a model.

<table>
<thead>
<tr>
<th>Impact on previously generated models</th>
<th>Efficiency</th>
<th>Traceability</th>
<th>Execution Context</th>
<th>Transformation engines</th>
<th>Independence</th>
<th>Transformation of non-involved elements</th>
<th>Information availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>re-transformation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Heerden et al., 2006)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>in live transformation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Heerden et al., 2006)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>coarse-grained</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(van der Meij, 2009)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fine-grained</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(van der Meij, 2009)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>direct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Comparison of Incremental Approaches
Performing Incremental Generation of Code in Model-driven Engineering

Among the implementations of MDE frameworks, the one which has been greater success in both business and academia is the Eclipse Modeling Project (EMP) (Gronback, 2009), located within the Eclipse (http://www.eclipse.org/) and in which there are implemented many of the specifications promoted by the Object-Management Group (OMG) (http://www.omg.org/), particularly those having to do with the MDA specification. In EMP, the meta-metamodel used is called Ecore, so close to the industry standard called Meta-Object Facility (MOF) (OMG, 2005a), that is taken as the core implementation of MOF, the Essential MOF (EMOF). Authors such us Gruschko et al (2007) justify the use of Ecore saying that it is the de facto standard in the industry.

The core of the Eclipse Modeling Project is the Eclipse Modeling Framework (EMF) (Steinberg et al, 2009). The EMF Compare tool (Toulmé, 2007), according to its authors, brings model comparison to the EMF framework, and provides generic support for any kind of metamodel in order to compare and merge models. The objectives are to provide a stable and efficient generic implementation of model comparison and to provide an extensible framework for specific needs. Furthermore, the output obtained is based on models, which provides the ability to further manipulations in an easy way through the use of a number of available projects and standard widespread software such as the EMP, widely used by scientists and companies. Brun and Pierantonio (2008) commented that EMF Compare was developed in 2006 to meet the need for a model comparison engine within the Eclipse Platform. Because of the capabilities and possibilities of this tool:

- It is integrated into the EMP.
- It supports any Ecore metamodel.
- It incorporates predefined heuristics to compare models.
- It offers extension mechanisms, we have chosen it to use in our prototype and case study.

EMF Compare allows obtaining the delta between two models through both graphical and programmatic interfaces. The programmatic interface is more powerful because it allows getting complete feedback using Java objects. Such feedback details the elements that have not changed between model versions (that is, the intersection) and the elements that have changed (that is, the delta). The sum of both gives the UDM value, which is the only artifact that contains all the information about additions, deletions, or changes in the source model. Together with the possibility of adapting the algorithms to the semantic of any metamodel, makes EMF Compare completely meets our requirements. Moreover, we also have chosen EMF Compare in a parallel project as the basis for checking a tool for testing match algorithms.

3.3 Representing the UDM

By representing the delta between two versions of a model, we can, at least in part, find out which parts of the model must be taken into account to regenerate artifacts because in most cases, it is not necessary to regenerate all of them. This is because information in the new model would exactly regenerate the same artifacts as in the initial model, resulting in wasted time and resources. Furthermore, using also the intersection among different versions of a model, we can obtain all the information required to regenerate any artifact from changes in the source model.

Moreover, to maintain the basic idea and retain the benefits of using a model-driven approach, we have a representation of the UDM based on the representation of differences between models proposed by Cicchetti et al (2007b) (Figure 4). The main objective is to always keep the last two versions of the model that have been introduced previously in some kind of repository. A key step is to extend the original metamodel in order to create a metamodel with the new metaclasses.
However, the presented difference model deviates from the original proposal (Cicchetti et al., 2007b) because if a change in one element causes artifacts to be regenerated from another element that has not changed; this element will not appear in the difference model and thus would cause inconsistencies. The proposed solution is to combine the difference model and the unchanged components of the latest version of the model, resulting in the representation of the UDM. To facilitate the use of existing technologies to create artifacts, we work with a model-independent difference (Konemann, 2009) which is self-contained; that is, the modifications are described without referring to external models.

Although the difference model could generate artifacts in many cases because it is based on metamodels and models, the representation of the UDM works in all the cases because it contains all the possible alternatives. Figure 6 shows an example based on the work of Cicchetti et al. (2007a), in which the difference between two Petri Nets (Peterson, 1981) models can be seen. At the top is the difference model. At the bottom are the additions needed in order to obtain the UDM. Thus, we have a new model, which is an instance of a metamodel derived from the original. The new model contains all the information needed to modify, add, delete or retain any of the elements involved in the system. In this example, the differences affect almost the entire model, and so we only need to add a few elements. However, in real-life scenarios, which generally imply much larger models, the differences are likely to affect only a very small model fragment.

### 3.4 Prototype Overview

On the basis of the proposed solution, we created a prototype that allows us to apply our solution to a real-world application. Figure 7 shows a general outline and then we discuss the main points of the system.
Performing Incremental Generation of Code in Model-driven Engineering

Step 1. Creating Standard Metamodels with Ecore

MDE initiatives such as MDA (Miller et al., 2003) or TALISMAN MDE (TMDE) (García-Díaz et al., 2010) advocate the generation of artifacts from a standard-based meta-metamodel (i.e., MOF). As the more commonly accepted implementation of MOF is Ecore, an Ecore metamodel must be designed as the first step in the incremental generation. This design will depend on the DSL used for each specific case. Once the metamodel is created, we can use any tool that supports creating and manipulating Ecore models.

Step 2. Extending the Metamodel with the New Data needed to Perform the Generation

The extension of the original metamodel is performed using the Ecore metamodel. This extension is derived during the development of the original metamodel. Note that this is not part of the generation of applications, and as such, this extension must be derived only once for each product family (Clements and Northrop, 2002). To extend the metamodel and to obtain the extended metamodel (Figure 4), we created a tool called MetamodelExt that creates the extended metamodel from an Ecore metamodel, in which four new metaclasses are added (i.e., added, deleted, changed, and unchanged) to each original metaclass, which are inherited from the original model. The MetamodelExt tool is part of the extension that we have developed called MWE Extends, which is a Modeling Workflow Engine (MWE) (http://www.eclipse.org/modeling/emft/?project=mwe) plug-in that supports incremental generation of code.

Figure 7: Proposed Architecture

Verion Control System

EMF Compare

UnifiedDiff

Xpand templates

Incremental software
Performing Incremental Generation of Code in Model-driven Engineering

Step 3. Creating the Unified Difference Model to know the Differences between Models

Unlike previous steps, this step takes place every time artifacts are generated. This is because it draws upon the differences between the present model version and the last version used. Thus, we can simultaneously manage the evolution of the model and the evolution of the artifacts that are generated from the model. To calculate the UDM, MWE.Extends includes the UnifiedDiff tool that calculates the difference between two models that conform to the same metamodel. That difference is actually a model that conforms to an extended metamodel created from the original one, which formally describes changes, deletions, and additions in the models through new metaclasses based on the metaclasses included in the original metamodel. The extended metamodel is created with the MetamodelExt tool. It also includes all the elements that remain intact in the latest version of the model to allow the generation of artifacts accordingly if necessary. The models arising from the differences will be first-class artifacts in the build process, which allows a single file to capture all the elements that have been changed, deleted, added, or left intact. Thus, we have all the necessary information in a single model, which allows the use of current technologies with respect to tools such as repositories, algorithms for comparison, generation templates, and so on.

Step 4. Generating the Final Artifacts in an Incremental Way

Once we have the UDM, the last step is to generate the artifacts using the extended metamodel. Instead of developing from scratch a model-driven tool for the incremental generation of artifacts, we have decided to adapt existing tools, particularly some of those developed under the EMP. The main reasons are that they are very close to the Model-Driven Architecture (MDA) standard, and they comply with the Eclipse Public License (http://www.eclipse.org/legal/epl-v10.html).

There are a lot of tools that could be valid for the generation of artifacts from a model. For instance, Java Emitter Templates, MOFScript, Velocity Template Language, Acceleo or Xpand (Rentschler and Becker, 2006). Our choice has been Xpand, a templating engine that supports the use, in a very easy way, of features such as polymorphism, which reduces the amount of extra code needed to manage the generation alternatives. This feature is essential to carry out a specific generation of artifacts for each of the possibilities with each model element (add, remove, change, unchange) in the most effective way.

To show this feature we can think of an example. A hypothetical metaclass called Person, which for each instance generates a class in the Java programming language. The designers of the metamodel could have decided that when a class is incrementally added in a model, it generates exactly the same code as for the initial case in which the model is still empty. In this case, the template for AddedPerson could inherit all the code written for the Person template, achieving the desired behaviour without the need of writing or duplicating more code in the template.

4. Case Study

To test and show the results provided by our tool, we created a case study based on a real application called Tweeterati (http://tweeterati.codeplex.com/), which is a Twitter client (https://twitter.com/). The Tweeterati application is composed of libraries that provide functionality.

However, as the application does not provide low-level configuration mechanisms, we built a DSL such that the source code can be changed in terms of programs or models created with the
Performing Incremental Generation of Code in Model-driven Engineering

DSL. Some of the features of Tweeterati, which can be modified by the DSL are:

- Add or remove search tabs.
- Change the messages shown to the user.
- Change how contacts are shown to users.
- Change how messages are shown to users.
- Change the meta-information of the software assemblies.

The idea is that if we use models from which Tweeterati artifacts are generated incrementally, we would get benefits that otherwise would not have been achieved because then we would always have to stop the application, make the pertinent modifications, and finally redeploy it. Thus, we did not have to recompile and deploy all the code in each case.

To carry out this evaluation, sequential changes in the input model were made, causing an evolution in the model in order to generate artifacts from the templates with polymorphic capacities. In total, we have made a total of 20 changes using different combinations of configurations to generate artifacts incrementally (e.g., to change the welcome message after changing the meaning of the commands of the application). The changes were adapted to both the type of application and the metamodel that was created for such an application. Note that it is only a simple case study used to show the possibilities of the approach. From here, it is possible to use the approach to any software and to any metamodel that corresponds to such a software, so the proposal is scalable for any type of project and can be adapted to the requirements of any particular case depending on what artifacts need to be created in an incremental way.

The generations that unfolded in response to changes in the model have yielded a series of results that suggest various benefits of using incremental generation of artifacts, especially if the results are extrapolated to other domains (Table 2).

Figure 8 shows, for space reasons, only a small fragment of the process, which can be sufficient to explain the way of working. Initially, in a part of the model (upper left), we can see that it has been specified that the words Games and Movies are searched in two search tabs. However, a change in the model (upper right) has led to stop searching Movies and start searching the keyword Music. In the centre, it is shown the UDM that corresponds to the evolution of the extracts of the modes, in which we can see how they are taken into account in a single model all the aspects that

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Traditional generation (always generating everything)</th>
<th>Incremental generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does it require stopping the application?</td>
<td>Yes</td>
<td>If it affects the main assembly</td>
</tr>
<tr>
<td>Is it possible conduct traceability?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Number of artifacts</td>
<td>1620</td>
<td>56</td>
</tr>
<tr>
<td>Size of artifacts (KB)</td>
<td>19.956</td>
<td>0.091</td>
</tr>
<tr>
<td>Generation time (seconds)</td>
<td>11.940</td>
<td>8.821</td>
</tr>
<tr>
<td>Deployment time (minutes)</td>
<td>2.905</td>
<td>1.313</td>
</tr>
</tbody>
</table>

Table 2: Results Obtained
Performing Incremental Generation of Code in Model-driven Engineering

can be interesting when incremental artifacts are generated. The bottom of the figure shows the UDM representation, basis for the next incremental generation of artifacts. The representation uses the XMI standard format (OMG, 2005b), which facilitates the use of different tools to generate artifacts. The figure also shows the skeletons of the templates made with Xpand, which can generate artifacts in terms of the different elements of the model. A great advantage of Xpand is that polymorphisms can be used, allowing the reuse of the previously created templates. For example, if we had previously created a template to generate artifacts each time a SearchElement item appears, and if with the incremental generation of artifacts we want to generate the same artifacts from all the ChangedSearchElement and AddedSearchElement elements as for SearchElement, we would only have to specify the template inheritance, reusing all the previously written code.

In this case study, major benefits are obtained with respect to the number and size of the artifacts to be generated. Generation time has also been reduced; however, this reduction is limited by the extra calculation needed to work with the extended metamodel and the UDM. In this particular
Performing Incremental Generation of Code in Model-driven Engineering

case, the gain in deployment time, although considerable, has not been very high because of the large number of dependencies among the files of the software solution. This is because they have been compiled and uploaded to a server during the deployment phase. However, in applications with fewer dependencies, the gain would be much higher. The same applies for applications which are composed of assemblies, as each of them could be compiled independently.

- The same results could not have been obtained with other approaches, for example:
  - The re-transformation approach only supports incremental development in the source but not in the target.
  - The live model transformation would force to create or modify a transformation engine on an ad-hoc basis and maintain the execution context that is susceptible to being lost when multiple users work concurrently across a network.
  - The fine-grained transformation would only regenerate artifacts from elements of the difference model.

The coarse-grained transformation would not have been carried out satisfactorily with the usual tools as they are designed to work with self-contained artifacts (Konemann, 2009).

5. Discussion

Without incremental generation, we must regenerate all the artifacts whenever there is a change in the input model. Incremental generation increases the likelihood that artifacts that are generated do not affect others, making it possible to change aspects of an application without stopping it. A typical example would be a model that generates SQL INSERT sentences. With an incremental proposal, we could generate an UPDATE sentence when the system detects a change in the model, thus updating the database at runtime. Otherwise, we must stop the application, remove the previous sentences, and make new INSERT sentences. Another example would be any Web application that could be modified without stopping it, or stopping only a component, without the need of uploading all the artifacts to a server each time someone makes a modification in the model that leads to the application.

One noteworthy issue is traceability. Assume that M, M’, and M’’ are different versions of the model in Figure 1 and that Δ, Δ’, and Δ’’ are the UDM from one version over the previous version, with as the empty set. In addition, let g(Δ), g(Δ’), and g(Δ’’) be the resulting generation of artifacts A, A’, and A’’, respectively. Then:

\[ M - ∅ = Δ = M \Rightarrow g(Δ) = A \]
\[ M’ - M = Δ’ \Rightarrow g(Δ’) = A’ \]
\[ M’’ - M’ = Δ’’ \Rightarrow g(Δ’’) = A’’ \]

Note that it is possible to know the different versions of a system and exactly what artifacts are created, deleted, or changed with regard to each of them. However, without incremental generation, traceability is not possible.

Regarding the number and size of artifacts to be regenerated both directly or indirectly, we note that incremental generation potentially reduces the number of artifacts generated and therefore also their size.

Generation time is the least important of the studied parameters because it does not directly affect the applications already deployed. Moreover, the incorporation of different algorithms and techniques for improving performance can provoke different generation times (e.g., in the
calculation or representation of differences between models). Furthermore, as both the number and the size of artifacts are potentially reduced, the deployment time is expected to be reduced as well.

6. Future Work

Future work should continue to evaluate and to improve the incremental generation of artifacts. An interesting issue in this context is the relationship between metamodel design and the artifacts that are generated. We hope to find relationships as direct as possible in order to avoid dependencies that can cause artifacts to be regenerated from a model element that has not undergone any alteration.

Also noteworthy is the application of the work proposed here to development environments, where the model view can be synchronized with the code view. Although many tools provide support for synchronization, this is usually linked to a development environment, a metamodel or, a specific architecture, but none of which uses a generic technique as outlined in this paper.

It is also interesting to study how the evolution, not only of the model, but of the metamodel (Garcés et al., 2009) may influence the incremental generation of artifacts. First, the models should be adapted to changes in the metamodel, and then they would be adapted to generate artifacts in terms of the modifications.

Finally, as the calculation of the differences between models is a key aspect to determine the artifacts that are generated incrementally from two versions of the same model, we want to mention that we are currently working on a tool called MCTest to study and evaluate model matching algorithms.

7. References


Performing Incremental Generation of Code in Model-driven Engineering


Performing Incremental Generation of Code in Model-driven Engineering

Performing Incremental Generation of Code in Model-driven Engineering

Biographical Notes

Vicente García Díaz is an associate professor in the Computer Science Department of the University of Oviedo. He has a PhD from the University of Oviedo in computer engineering. His research interests include model-driven engineering, domain-specific languages, project risk management, software development processes and practices. He has graduated in Prevention of Occupational Risks and is a Certified Associate in Project Management through the Project Management Institute.

Begoña Cristina Pelayo García-Bustelo is a lecturer in the Computer Science Department of the University of Oviedo. She has a PhD from the University of Oviedo in computer engineering. Her research interests include object-oriented technology, web engineering, eGovernment, modeling software with BPM, DSL and MDA.

Oscar Sanjuán Martínez is a lecturer in the Computer Science Department of the Carlos III University of Madrid. He has a PhD from the Pontifical University of Salamanca in computer engineering. His research interests include object-oriented technology, web engineering, software agents, modeling software with BPM, DSL and MDA.

Juan Manuel Cueva Lovelle became a mining engineer from Oviedo Mining Engineers Technical School in 1983 (Oviedo University, Spain). He has a PhD from Madrid Polytechnic University, Spain (1990). From 1985 he has been a professor at the languages and computers systems area in Oviedo University (Spain), and is an ACM and IEEE voting member. His research interests include object-oriented technology, language processors, human-computer interface, web engineering, modeling software with BPM, DSL and MDA.