Staged Model-Driven Generators
Shifting Responsibility for Code Emission to Embedded Metaprograms

Yannis Lilis[1], Anthony Savidis[1, 2] and Yannis Valsamakis[1]
[1]*Institute of Computer Science, FORTH, Heraklion, Crete, Greece*
[2]*Department of Computer Science, University of Crete, Crete, Greece*

Keywords: Model-Driven Engineering, Multistage Languages, Code Generation, Compile-Time Metaprogramming.

Abstract: We focus on MDE tools generating source code, entire or partial, providing a basis for programmers to introduce custom system refinements and extensions. The latter may introduce two maintenance issues once code is freely edited: (i) if source tags are affected model reconstruction is broken; and (ii) code inserted without special tags is overwritten on regeneration. Additionally, little progress has been made in combining sources whose code originates from multiple generative tools. To address these issues we propose an alternative path. Instead of generating code MDE tools generate source fragments as abstract syntax trees (ASTs). Then, programmers deploy metaprogramming to manipulate, combine and insert code on-demand from ASTs with calls resembling macro invocations. The latter shifts responsibility for source code emission from MDE tools to embedded metaprograms and enables programmers control where the produced code is inserted and integrated. Moreover, it supports source regeneration and model reconstruction causing no maintenance issues since MDE tools produce non-editable ASTs. We validate our proposition with case studies involving a user-interface builder and a general purpose modeling tool.

1 INTRODUCTION

In general, Model-Driven Engineering (MDE) involves tools, models, processes, methods and algorithms addressing the demanding problem of design-first system engineering. An important authoring requirement for such tools is to involve notions and concerns inherent in the design domain. In this context, either general-purpose notations are adopted in software modeling, or mission-specific models are offered for very specific tasks. Then, target implementations are incrementally derived, usually with various intermediate transitions from the modeling to the implementation domain, until eventually reaching a source code representation.

The latter relates to generative MDE tools that deploy code generators to automatically produce source code for the various modeled entities. The generated code may contain special tags carrying model information in order to allow model reconstruction, while it is typically extended with custom user code to deliver the final application. The primary tool-chain of generative MDE frameworks is outlined under Figure 1.

Our work falls in the field of generative model-driven tools and focuses on addressing the maintenance issues arising from code generation. We continue elaborating on parameters of the problem and then brief the key contributions of our work to address this issue.

1.1 Problem Definition

MDE tools cannot optimally address all required features of an application at the software engineering level. As a result, custom source code amendments and modifications are always anticipated. Even if advanced methods are deployed to modularize and decouple generated code from custom application code, one can never exclude the possibility that interdependencies or custom updates may appear.

The typical lifecycle of the generated code is outlined under Figure 2. As shown, a dependency is introduced by having the application logic directly refer and deploy generated components (middle part). But for most languages this is overall insufficient for effectively linking application and generated code, practically requiring the generated code to be also manually modified. Typical updates relate to application functionality importing and

Lilis Y., Savidis A. and Valsamakis Y..
Staged Model-Driven Generators - Shifting Responsibility for Code Emission to Embedded Metaprograms.
DOI: 10.5220/0004878605090521
In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2014), pages 509-521
ISBN: 978-989-758-007-9
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

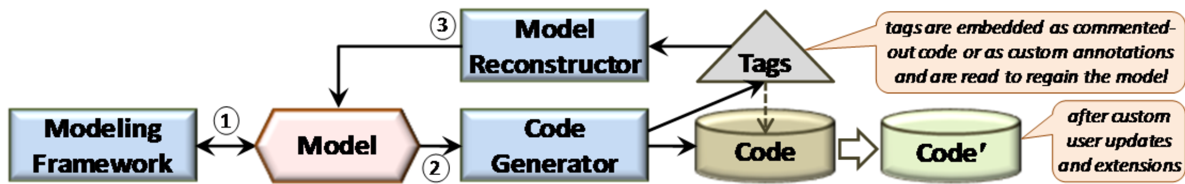


Figure 1: Tool chain in generative MDE tools: (1) interactive model editing; (2) code generation from a model; and (3) model reconstruction from tags inserted in the generated source code, carrying model information.

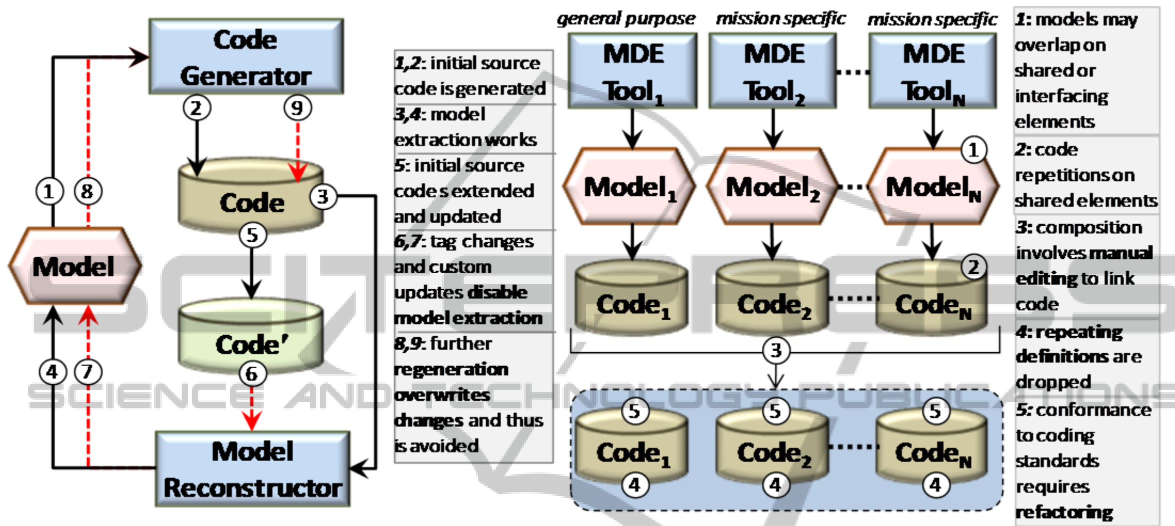


Figure 3: Maintenance issues involved in the deployment of generative MDE tools, individually (left) or collectively (right).

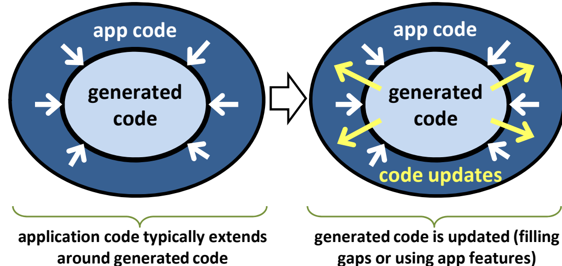


Figure 2: Common growth of application code around the originally generated code; future custom extensions and updates eventually lead to bidirectional dependencies.

invoking, application-specific event handling, linkage to third-party libraries that are not known to the MDE tool, code improvement or refactoring. This situation very quickly results into many bidirectional dependencies (right part).

The latter maintenance issues are detailed in the typical generative model-driven process shown in Figure 3. Initially, if the code is not changed, source regeneration and model reconstruction are well-defined (left, steps 1-4). In other words, the MDE tool works perfectly for both steps of the processing loop. However, once the generated code is updated (left, step 5), two problems directly appear. Firstly, tag editing and misplacing may break model reconstruction (left, steps 6-7), while any code manually inserted outside the MDE tool causes a model-implementation conflict. Secondly, source regeneration overwrites all manually introduced updates (left, steps 8-9). For real-life applications of a considerable scale the latter may lead to the adoption of the MDE tool only for the first version, or worse, avoiding using an MDE tool at all.

Maintenance issues also arise when trying to combine the outcome of multiple MDE tools. When using multiple tools, a single application element may end up being shared by different models. This means that when the code for each model is generated, there will be code repetitions for the shared elements (right, steps 1-2). In this case, the developer has to manually edit the generated sources to drop any repeated definitions and link the code properly (right, steps 3-4). Furthermore, the use of different MDE tools implies different code generators and thus different coding styles and methods present in the generated code. Having all generated sources conform to specific coding standards inevitably requires manual refactoring (right, step 5).

1.2 Contributions

Our main contribution is an inversed responsibility model for generative MDE tools (Figure 4) where: (i) the code for model entities becomes available in the form of ASTs; and (ii) the actual code generation is applied *on-demand* and *in-place* through generative macros embedded in the main program as a metaprogram that is evaluated before compilation.

Figure 4: Outline of the proposed MDE approach.

This approach addresses the maintenance issues of traditional generators and sets code manipulation as a first-class concept in the MDE, revealing the value of using a metaprogramming language in this context. Apart from compile-time evaluation, we also elaborate on how our approach can be applied using the runtime metaprogramming facilities of languages like Java and C#.

Overall, we propose an improved process where the MDE tool outcome is read-only, decoupled from source code generation, letting the application directly deploy and manipulate generated code fragments, instead of being built around them. In this context, we also discuss how AST composition allows combining sources whose code originates from multiple MDE tools.

2 RELATED WORK

Our work targets the field of MDE and focuses on the maintenance issues arising from source code generation. In this context, we review advanced and popular general-purpose generative MDE tools.

The *EMF* (Eclipse Foundation, 2008) project is a modeling framework and code generation facility for building applications based on a structured data model. The model is described in the *Ecore* meta-model, while code generation targets Java and uses *@generated* annotations to specify the automatically generated code segments. By default, all generated code segments include this annotation and are overwritten upon regeneration. In case the generated code is manually extended, the *@generated* annotations should be manually removed (or alternatively turned into *@generated not*) to specify that the annotated code segments should be maintained and not overwritten upon regeneration. However, manual extensions cannot be reflected back to the model while model updates will be discarded for manually extended code. Additionally, misplacing or forgetting to remove the annotations may result in losing manually written source code.

Acceleo (Obeo, 2006) is a code generator implementing the OMG's Model-To-Text-Language specification. It is independent from the targeted technology allowing the generation of any textual format using plugins while it provides an OCL-oriented (OMG, 2012) template-like definition for expressing custom generators. Acceleo supports incremental generation allowing developers to regenerate target files without losing modifications once explicit *[protected] … [/protected]* constructs are used, translated into tagged comments that mark a code region that will not be overwritten during regeneration. Again, any intervention on such tags breaks regeneration. Furthermore, placing such tags requires an a priori knowledge of the locations requiring manual updates, something not always available during the design phase. Practically, this means that for each required update, the developer will have to go to the transformation script, insert a protected code region, regenerate the code and finally go back to the source to perform the update. This is a tedious circle, while in our approach updates are normally applied on the edited source with no such context switches or regeneration.

Actifsource (Actifsource GmbH, 2010) is a design and code generator tool focusing on domain-driven software development. It utilizes a template-based code generation approach including by default various language generator templates, while allowing new ones to be added for any language. Like Acceleo, Actifsource also supports using special tags to specify protected regions where manually inserted code will not be overwritten upon regeneration. Again, any intervention on these tags causes maintenance issues once code is regenerated.

Umple (Badreddin and Lethbridge, 2013) is a modeling tool that reduces the distance between model and code by adopting UML abstractions directly into a high-level programming language code. This way, models become just another abstract code view, eliminating the need for model extraction

as everything in the model is represented directly in code. Umple can generate code for languages like Java and PHP and allows embedding native code or transforming the generated code through aspect-oriented facilities. Umple's philosophy for generated code is that it should never be edited but treated as a development artifact that can be thrown away and recreated and thus, there is no issue of round-tripping (Antkiewicz, 2007; Chalabine and Kessler, 2007). Our approach, maintains the separation between model and code while overcoming the round-trip issue through the in-place deployment of code fragments generated by the model.

Papyrus (Lanusse *et al*., 2009) and *Modelio* (Desfray, 2009) are both MDE tools offering code generation for a many languages. They support the full MDE development cycle allowing both model-to-source and source-to-model transformations. For the latter, they parse source files locating specific code structures (e.g. classes, attributes, operations etc.) in order to regenerate the model, while treating any additional code they include as metadata. This is an important step towards resolving the maintenance issues; however, it cannot be applied in case the generated code originates from multiple models. Also, such a reverse engineering policy is valid for general-purpose MDE tools but cannot be deployed for mission specific tools. For example, in case of MDE tools for user-interface code generation, like *GuiBuilder* (Sauer and Engels, 2007) or *GrafiXML* (Michotte and Vanderdonckt, 2008), it is practically impossible to recognize the widget elements by parsing manually written source code (Staiger, 2007). Our methodology can be deployed for both general-purpose and mission-specific tools, while still addressing the maintenance issues.

Maintaining manual updates after regeneration is also possible by adopting the Generation Gap Pattern (Vlissides, 1996). For instance, in Xtext/Xtend (Bettini, 2013) the generated code is placed in a separate source folder *src-gen*, whose contents are overwritten and should thus never be modified. On the first generation, Xtext also generates stub classes in the normal source folder that inherit from class in the *src-gen* folder. These classes are never regenerated and can thus safely be edited without the risk of being overwritten. However, this approach does not work well when generated classes are involved in existing class hierarchies, while it also complicates system design. Additionally, it does not address the issue of combining generated code from multiple MDE tools.

Our proposition for improving the MDE process involves metaprogramming techniques. In this context, we also consider work utilizing generative programming and aspect-oriented programming techniques for MDE to be related to ours. Völter and Groher (2007) explore aspect-oriented techniques for model-driven code generation, while Hemel *et al*. (2010) and Zschaler & Rashid (2011) treat source code generation as another model transformation utilize a rewriting-based technique to compose and combine partial generation represented respectively in AST and text form. While in all cases code generation is achieved, the final source may be freely edited with no additional consideration for the involved maintenance issues.

3 STAGED METAPROGRAMS

Generally, metaprogramming relates to functions which generate code, i.e. *programs producing other programs*, while metaprogramming languages take the task of code generation and support it as a first-class language feature. This is a sort of reification of the language code generator enabling programmers to write code which generates extra source code. When available as a macro system before compilation, the method is known as *compile-time metaprogramming*. Alternatively, if offered during runtime – typically using the language reflection mechanism – it is called *runtime metaprogramming*. We focus on compile-time metaprogramming as it is more powerful than its runtime case. In this context, code generating macros are functions manipulating code in the form of ASTs, and are evaluated by a separate stage preceding normal compilation. Then, they are substituted in the source text by the code they actually produce. Due to the introduction of an extra stage, and because macros may generate further macros, thus requiring extra staging, such languages are also called *multistage languages* (Sheard *et al*., 2000; Taha, 2004). In our work we use the dynamic object-object language Delta (Savidis, 2012), along with its compile-time metaprogramming extension (Lilis and Savidis, 2012). Popular meta-languages also include *Lisp* (Steele, 1990), *Scheme* (Dybvig, 2009), *MetaML* (Sheard, 1999), *MetaOCaml* (Calcagno *et al*., 2001), Template Haskell (Sheard, 2002), *MetaLua* (Fleutot, 2007) and *Converge* (Tratt, 2005).

In the Delta language, meta-code involves meta definitions and inline directives (i.e., code generation), prefixed with the **&** and **!** symbols respectively. In particular, inline directives accept an expression returning an AST and are the only way to insert extra code into the main program.

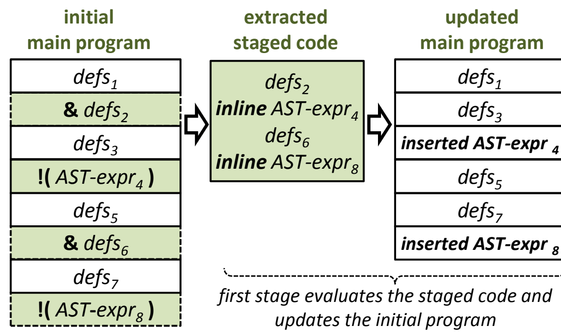


Figure 5: Evaluating generative macros with an extra stage.

As shown in Figure 5, during the first stage the compiler: (i) collects all scattered meta-code into a single metaprogram; (ii) evaluates the program while recording the output of the inline calls; and (iii) removes all meta-code from the initial program and replaces inline directives by the code they produced. For example, consider the following code.

```
using std;
&ast = load_ast("<ast path>");
!(ast); ← code generation directive
```

The first line is normal code, a directive to import standard library functions. But the next two lines are meta-code, indicated by **&** and **!** prefixes. The second line loads an AST from a file, assume the loaded AST to be the one of Figure 6. The third line inserts the code implied by this AST into the main program. As a result, *after* the first stage, and *before* normal compilation, the main program is:

```
using std;
function square(x) { return x * x; }
print(square(2));
```

Such code is only transient, and exists inside the compiler temporarily during the first compilation stage. It is shown here for clarity. After this first stage, the resulting source text constitutes the input to the normal compilation phase, *as if it was originally written this way* by the programmer.

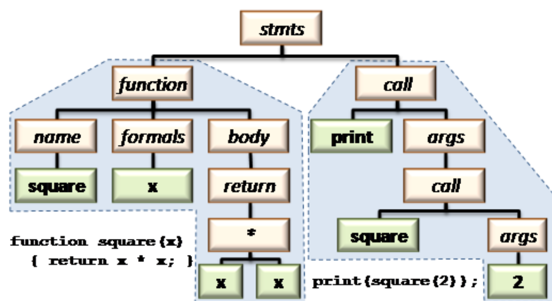


Figure 6: Abstract syntax tree example.

The previous example shows only the creation and inlining of an AST value. However, metaprograms typically operate on AST values, adding, removing or transforming nodes they contain. For example, consider that we wanted to extend the above code with an extra print statement. To achieve this, we would have to obtain and manipulate the children of the root *stmts* node:

```
&ast = load_ast("<ast path>");
&stmts = ast.get_children();
&stmts.push_back(<<print("...")>>);
!(ast); ← generate transformed code
```

The notation <<...>> is not a conceptual symbolism, but actual Delta syntax relating to a meta-language construct known as quasi-quoting. Essentially, it is a compile-time operator that converts the surrounded raw source-text to its respective AST representation. For instance *<<1+2>>* is equivalent to the AST of the expression *1+2*, not just the character string *'1+2'*.

Performing transformations on ASTs requires knowledge of the particular AST structure as well as information about the language AST representations. Essentially, locating nodes that should be transformed may involve tedious traversal of the AST. This can be improved using a decoration process that allows direct navigation across AST nodes through *named* entities involved in the AST structure. In the above example, inserting code within the body of the *square* function would be achieved with the following meta-code statement:

```
&ast.square.body.insert(<<...>>);
```

This way, high-level knowledge of the AST contents and a simple tree manipulation API are sufficient for introducing elaborate AST extensions.

4 PROPOSED PROCESS

The primary motivation for our work has been the serious source code maintenance issue inherent in the deployment of generative MDE tools. Although we needed to avoid this problem, in the mean time we wished to retain all powerful features of generative MDE tools. Thus we started thinking of an alternative path, in which: (i) the MDE tool output would somehow remain invariant, that is in a not-editable form; and (ii) the source code of the application could still grow and evolve in an unconstrained manner around it. This led us to the idea of bringing staged metaprogramming into the pipeline by enabling programmers algorithmically

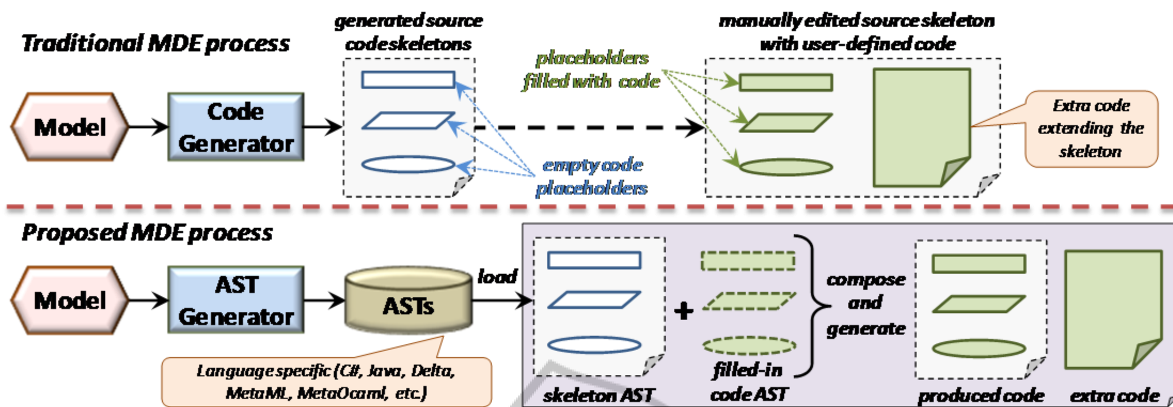


Figure 7: Top: Traditional MDE process where the generated source code files are manually updated with fill-in and extra code. Bottom: The proposed MDE process where the tool output is in AST form and the programmer deploys embedded metaprograms to load, compose and generate the model code that is integrated along with the custom application code.

manipulate the generated code including: loading, processing and transforming. We continue detailing our proposition for a refined model-driven process and compare it against the traditional process. We also discuss how a similar approach could be deployed in languages that do not offer explicit support for staging but support some degree of runtime composition though a reflection API.

4.1 Refined Tool Chain

Our proposition for a refined model-driven process is deployed on languages with explicit support for staging and could be directly applied on all multistage languages discussed earlier. We utilize two stages of evaluation, one for the evaluation of the generative macros and another one for the normal program translation. In particular, with staged code generation, the MDE process, outlined under Figure 7 in analogy to the traditional generative process, is improved as follows.

Initially, the model-driven tools generate code in the form of language-specific ASTs. Apart from code, the ASTs can also incorporate any special code annotations, like those required by various Java frameworks. ASTs are essentially read-only data, meaning the result of the code generation remains unchanged and thus code-to-model reconstruction is unnecessary. Then, any custom application code, that would typically require manual extensions on top of the generated source code files, is instead developed as a full program that deploys embedded metaprograms to load and incorporate the model code as needed. Essentially, these metaprograms include functionality for reading and manipulating ASTs as previously discussed, so their evaluation can generate a transient model code version (in read-only form) that can directly incorporate custom application code. If any changes are performed on the model, then further regeneration overwrites only the ASTs and not the source file that contains the custom application code. This means that on the next translation the metaprograms will simply load the updated AST versions, generate the updated transient model code and then directly integrate it with the application code without requiring any additional actions from the programmer, effectively improving the maintenance of the system.

The adoption of an AST representation for model code also enables the combined deployment of multiple MDE tool outputs. Metaprograms can load and manipulate multiple ASTs regardless of the originating tool, so supporting such combination is just a matter of specifying the appropriate AST composition for the input models. Additionally,

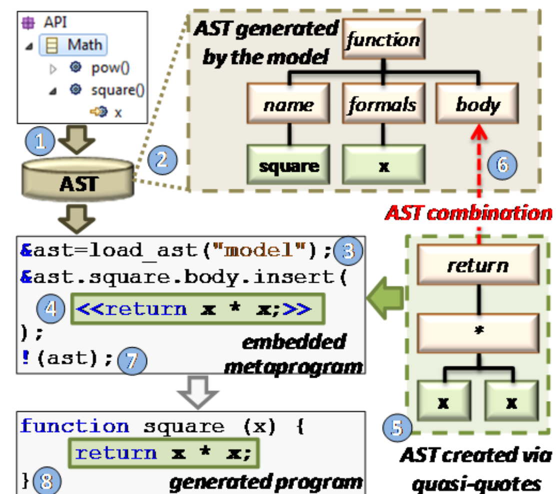


Figure 8: Example of deploying the proposed MDE process. Highlighted steps 1-8 are discussed within text.

metaprograms may contain any further application-specific composition or editing logic. This means that it is possible to perform any code transformation on a source fragment before inserting it in the final source. Finally, after the staged evaluation has produced the final source, the process continues with the normal program translation or evaluation.

To better illustrate the steps involved in the proposed process we discuss the deployment example of Figure 8. The scenario involves a model representing an API for mathematical functions where we want to automatically derive the function skeletons and then complete them with the implementation code. For brevity, we only show the implementation of a single function.

The model is initially passed to the AST generator to produce its AST code representation (step 1). The latter will contain the code skeleton for the definition of function *square*, and is stored as a data file (step 2). The programmer then develops the custom application code and uses an embedded metaprogram to integrate it with the model code. In particular, the metaprogram begins by loading the AST data (step 3) and then manipulating it to incorporate the custom application code directly into the loaded model code (step 4). The custom application code here is just the *return x*x;* statement that is turned into an AST through quasi-quotes (step 5). The call to *ast.square.body.insert* will then combine the two ASTs (step 6), resulting in a single AST with the fully implemented function. This AST is then inserted directly in the program source through a code generation directive (step 7), leading to the final source code version (step 8).

4.2 Comparing with Current Practices

We continue with a comparison between the proposed and the traditional process when model changes are involved. We reuse the model of mathematical functions, and suppose we want to extend it with further definitions.

In our approach, the application code specified by the programmer as well as other client code that relies on it, works with no issues and without involving any additional programmer intervention. Any existing functions for which implementations where originally specified will still be present in the new AST, so the metaprogram will combine them with their matching implementations as before. Newly introduced functions will simply be inserted with empty implementations. For them to be fully functional, the programmer should naturally provide their implementations explicitly.

A traditional MDE tool that blindly overwrites source code upon regeneration naturally requires additional actions to maintain the previously specified function implementations, involving a temporary copy of the source code and a manual code merge after the regeneration process.

More elaborate tools that allow custom extensions to be retained across regenerations, i.e. automatically merge manually updated code with new model code, yield better results but do not fully solve the problem. For instance, consider the previously discussed use of *@generated* annotations kept only on functions that should be overwritten and removed from functions with custom extensions. For the mathematical API example, this means that original functions that were implemented no longer have a @generated annotation. This way, when the API is extended, regeneration will introduce the new functions without overwriting the previous ones, achieving the desired functionality. However, consider a different model update involving modifications for already implemented functions, like adding an extra argument to some functions. Since original functions versions are maintained, the regeneration process introduces duplicate function skeletons with updated prototypes. The programmer should then manually move the implementations from the original bodies to the matching new ones, drop the old entries and finally specify that the new functions contain user code by removing their *@generated* annotation. Clearly, for multiple model updates or a large number of modeled entities this is a tedious and error-prone process.

Using our approach, such a model update requires no further actions and is handled as before: the updated model is loaded in AST form and then the function implementations are inserted where needed through AST manipulation without being affected by the newly introduced argument. Practically, the metaprogram specifies the logic for integrating custom application code directly within the model code, so as long as the model structure matches this insertion logic, no model updates break the regeneration process. Inconsistencies in the metaprogram can only occur if some model entities are removed, causing any meta-code that tries to access them to fail. Nevertheless, the same happens in traditional tools when previously generated code that is retained tries to access model entities that no longer exist, resulting into compilation errors.

4.3 Deploying using Reflection

Not all popular languages support staging, even

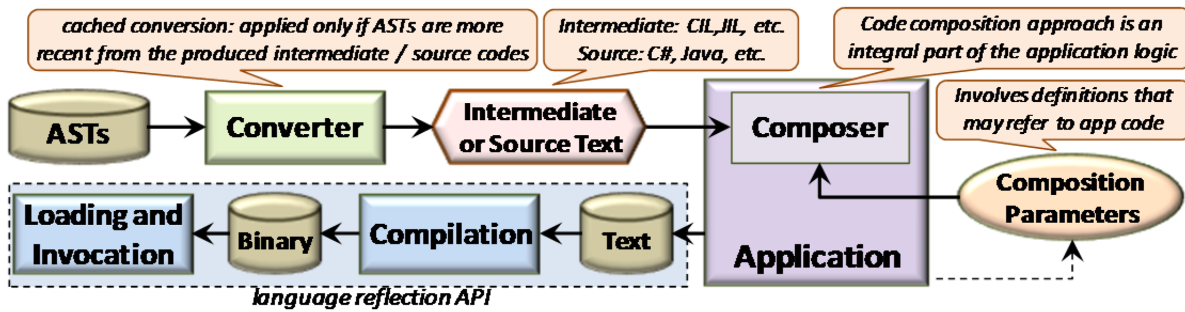


Figure 9: Applying the proposed generative MDE process without explicit staging; the application composes intermediate or source text and then deploys the language reflection API for compilation and invocation (JIL stands for Java Intermediate Language, CIL for the Common Intermediate Language of .NET). The entire runtime conversion, composition and compilation process is cached – it is only repeated when the ASTs change, i.e. upon regeneration.

though there are a few third party extensions such as Metaphor (Neverov and Roe, 2004) and Mint (Westbrook *et al.*, 2010). In this context, one may deploy the reflection mechanism of languages like C# or Java to practice a similar source code management and generation pipeline as the one discussed in the previous section. This option is detailed under Figure 9, showing that the language compiler and the dynamic class loading and method invocation facilities (i.e. reflection API) are directly deployed. The entire process starting the conversion from ASTs to intermediate representations (very flexible, suggested), or alternatively to source text (more rigid, not suggested), should be explicitly implemented as it is not automated by the languages. However, it is cached, meaning it is not repeated during execution, but applied once per AST version. The oval of Figure 9 labeled as *composition parameters* represents the need for performing custom mixing between the automatically generated source code and the manually inserted code, something that is apparent in the presence of *Composer* as an integral part of the application. This is similar to AST composition alternatives, although at the intermediate representation level, and is very critical to ensure that maximum code mixing freedom is provided to developers.

5 CASE STUDIES

To validate our approach and assess its expressive power and engineering validity, we have carried out two case studies with proof-of-concept prototypes, one focusing on user-interface code generation and another one creating an entire class hierarchy based on a given model. The goal of our studies was twofold: (i) to show that the maintenance issues are effectively eliminated; and (ii) to demonstrate the huge expressive power of metaprogramming for flexible code composition. The source code for the case studies along with a video demonstrating the entire MDE process is available at the Delta site (Savidis, 2012) under the metaprogramming section.

5.1 User Interface Modeling

We have adopted *wxFormBuilder* (2006), a popular publicly available interface builder for the *wxWidgets* cross-platform library. It offers a typical rapid-application development cycle with interactive user-interface construction, and outputs interface descriptions into its custom language-neutral format called XRC (XML Interface Resources). Using this tool, we modeled the interface of a paint application. To convert XRC data to the Delta language ASTs we built and deployed an appropriate AST generator. Then, using the metaprogramming features of the Delta language, we imported and manipulated the application ASTs, and also added extra interactive features and behavior to it, besides the ones introduced just with *wxFormBuilder*. Finally, we inserted custom extra code (e.g. event handling) to offer a fully-functional application.

The entire modeling and implementation process was incremental so as to involve multiple model updates. In particular, we began with a primitive user-interface model, consisting only of the canvas and the painting tools (Figure 10, top-left), generated the corresponding AST and provided the necessary source code implementation (Figure 10, middle). Then we gradually updated the user interface model to include additional toolbars (Figure 10, top-right). For each model update, we also provided the matching implementation code (Figure 10, bottom). Despite model updates, no maintenance issues arose during the entire process. Actually, after each model update, the previous source code version compiled

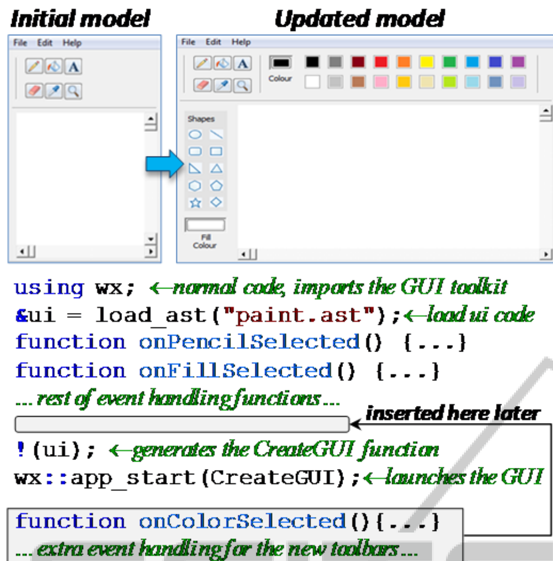


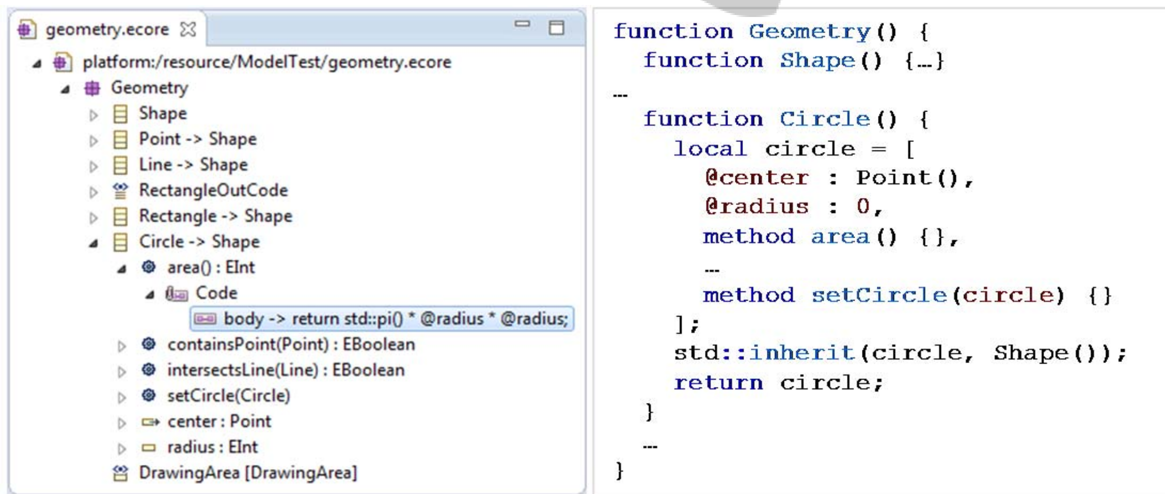
Figure 10: Deploying our MDE process for user-interface generation. Top: Initial and updated interface models; Middle: Original application code encompassing staged code; Bottom: Code extensions for the updated model.

and executed normally without any changes, while naturally offering no interaction for the newly introduced toolbar. Finally, implementing new functionality simply required inserting code in the source file where necessary.

5.2 Class Hierarchy Modeling

We used the Eclipse Modeling Framework (EMF) to model a class hierarchy for the core logic of a paint application. Among other things, the model included functionality for drawing shapes, containing classes like points, lines, circles, etc. The model was created through the Ecore meta-model and its specification was generated in XMI format. Then, as with XRC earlier, we implemented a custom generator from XMI to ASTs. Figure 11 shows the model in the EMF Editor, the respective code structure (shown as source text, although they are manipulated in AST form) as well as the deployment code required to inline the code AST in-place with the normal program code. Again during the process, we reloaded the model and regenerated the XMI specification to verify that no maintenance issues were introduced in the development process.

For the method implementations of the modeled classes we practiced two alternative methods. The first one involved specifying the method bodies directly in the model through the use of special EAnnotation elements (Figure 11 top-left, highlighted). The second one did not involve any




```
&classes = load_ast("geometry.ast");       ←load AST definitions generated by the XMI model
&classes.Geometry.Circle.area.body.insert(   ←insert custom code for generated methods
  <<return std::pi() * @radius * @radius;>>
);
...any other meta or normal code may be freely placed here ...
!(classes);                  ← inline the entire AST carrying the class hierarchy at this source location
```

Figure 11: Top-left: Ecore model of the target class hierarchy; Top-right: Code structure (AST) generated by the model; Bottom: Deployment code for loading the model code, performing manual updates through AST editing and inlining the final AST code. The initial value of the meta-variable classes corresponds to the code structure shown at top-right.

model editing, but relied on directly inserting code in the method bodies through AST manipulation as previously discussed. This approach may seem more difficult to adopt, but in fact it is easy to develop and offers several advantages over the first one.

When inserting the code directly in the model, the code is entered as raw text and thus lacks any programming facilities. Additionally, code overview is severely restricted, as the model view truncates the annotated text and full code inspection is only allowed for a single selected EAnnotation. Of course, there is no direct notion of parameterization or reuse; the only option short of code repetition is to explicitly introduce new model methods, implement their code through a new EAnnotation and use corresponding invocations where needed, again as raw text placed in other EAnnotations. In any case, inputting source code in separated text areas is far from a productive development method.

Regarding the second approach, creating or inserting code through metaprogramming is achieved through additional syntax (quasi-quotes) directly at code editing level. This means that the developer may utilize all typically offered code facilities like syntax highlighting, auto-completion, refactoring tools, etc. Additionally, different code segments (ASTs) corresponding to related methods or classes may be placed in the same source location as would be the case if the entire class was manually written by the developer, thus supporting the typical source code overview. Finally, since ASTs are actually metaprogram data, they are subject to standard software engineering practices like parameterization, encapsulation, composition, etc.

5.3 Combining Modeling Tools

The last step of our case study focused on obtaining the code generated by the previously discussed methods and combining it along with the custom application logic to implement a fully functional paint application. To emphasize the compositional flexibility of our proposed approach in combining independently authored interfaces under a single system, we further utilized two separate user-interface models, one for the main paint interface and another for the shapes toolbar extension.

A simple concatenation of the generated sources caused no direct compilation conflicts; however it was far from sufficient for deriving a fully-functional application. In fact, many manual updates were necessary for both generated components, some of them involving bidirectional dependencies. First, the two interfaces had to be combined to a single one. Then, certain methods of the class hierarchy like *draw* required invoking UI-related operations. However, the class hierarchy model was unaware of the deployed UI library, meaning that such information could not be available in the model and would thus have to be explicitly expressed as a manual extension in the generated sources. Finally, we needed to combine the generated code with the custom application logic. The meta-code implementing the above functionality is outlined under Figure 12, with details removed for clarity.

We begin by loading the ASTs that were previously generated by the XRC interface definitions (both the paint UI and the shapes toolbar extension) and the XMI class hierarchy model (step

```
using wx;                                    ← normal code, directive for importing the wxWidgets GUI toolkit
---------------------------------------------------------------------------------------------------
&paintUI = load_ast("paint.ast");            ← load the AST of the paint application user-interface code
&shapesUI = load_ast("shapes.ast");          ← load the AST of the shapes toolbar user-interface code
&classes = load_ast("classes.ast");          ← load the AST of the class hierarchy for the toolset      (1)
---------------------------------------------------------------------------------------------------
&function MergeGUI(main, toolbar){…}         ← compile-time function to integrate an interface containing
&MergeGUI(paintUI, shapesUI);                ← a toolbar UI to the main program UI                       (2)
---------------------------------------------------------------------------------------------------
classes.Geometry.Circle.draw.body =          ← insert custom implementation for method Circle::draw(dc)
  <<dc.drawcircle(@center, @radius);>>;      ← dc: argument, @center and @radius: circle attributes
… other shape method implementations are inserted here as well…                                         (3)
---------------------------------------------------------------------------------------------------
… custom functionality and event handling code…
---------------------------------------------------------------------------------------------------
… any other meta-code or normal code may be freely inserted here…                                       (4)
!(classes);          ← inline the transformed classes AST at this source location
… any other meta- or normal code may be freely inserted here…
!(paintUI);          ← inline the transformed paintUI AST at this source location – generates function CreateGUI
… any other meta- or normal code may be freely inserted here…
---------------------------------------------------------------------------------------------------
wx::app_start(CreateGUI);  ← normal code, uses the generated CreateGUI function to launch the GUI
```

Figure 12: Meta-code to load, manipulate and inline the source code of all modeled aspects of our system. The result is a fully functional paint application like that shown on the top-right of Figure 7.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1). The interface definitions are combined as needed to generate the final application interface (step 2). In particular, the top level frame of the shapes toolbar is dropped and the remaining interface component (i.e. a panel) is inserted in the frame of the paint application before the canvas. Then, we implement the various methods of the class hierarchy (step 3) by creating and inserting AST values in the method bodies as previously discussed Notice that the quasi-quoted code can directly link to UI elements. Finally, once all appropriate transformations and extensions have been performed on the ASTs, they can be inlined to the final program at some source location (step 4). The AST of the class hierarchy should be inlined first so as to be available in the subsequent UI code that utilizes it. The code of the class hierarchy also requires the GUI toolkit functionality; however it is already visible through the import directive present in the first line.

6 DISCUSSION

Our approach overcomes the maintenance issues of generative MDE tools; however its deployment naturally involves some tradeoffs.

Firstly, it requires applying an advanced programming technique such as metaprogramming in an already demanding field like MDE, potentially leading to increased system complexity. For instance, creating and manipulating ASTs to perform code updates is arguably harder than manually editing the corresponding source code segments. Nevertheless, the use of quasi-quotes enables creating ASTs just like writing normal code, while AST manipulation can be simplified with better support for AST traversal (e.g. the name decoration process discussed earlier) along with a simple tree editing library.

Another issue concerns the transformation of the MDE tool output into an AST and requires a separate converter per deployment language as well as per model format. For instance, in our test cases we had to build two converters (one for XRC and another for XMI) to support the two modeling tools we used. Moreover, if we wanted to use our approach in another language we would have to create similar converters generating ASTs for that language. In a setup with varying languages and diverse model formats this arguably introduces an overhead in the MDE process. However, a single converter may be used for developing multiple applications that share a development language and a model format thus reducing the amortized effort required for a particular application. The effort required for such a converter is proportional to the complexity of the target model specification. Typically, it should be similar to creating a model-to-code transformation but with the output being the source code AST instead of the source code text. For MDE tools that already provide model-to-code transformations in the deployment language, an alternative requiring significantly less effort is to first use the transformation to get the generated sources, parse them into ASTs and finally manipulate them as needed (e.g. remove code segments not directly relevant to the modeled entities) to be ready for deployment. Additionally, it is possible to further reduce the effort required to implement a converter for a specific format across different languages. The converter may have a language-independent core handling the target format and utilize multiple language-dependent back-end plugins to support the various deployment languages. In this sense, all common converter functionality is only written once, thus minimizing the overhead of supporting additional languages.

7 CONCLUSIONS

Currently, model-driven engineering represents a domain of powerful development tools facilitating the modeling of systems and supporting the transformation process from abstract to concrete models, eventually down to the physical platform level. Generative MDE tools support the production of concrete application implementations directly at the source code level. Such a facility is very helpful, powerful and flexible for software development. However, it also causes maintenance issues once extensions and updates are manually introduced over the initially generated model code or when trying to combine sources coming from multiple MDE tools.

To cope with such maintenance issues we propose the exploitation of the metaprogramming language facilities and suggest an improved model-driven code of practice relying on the manipulation of source code fragments by clients directly as data. In this approach, the generator components of MDE tools need output ASTs, not source code, while clients should import and compose ASTs as needed, before eventually performing on-demand and in-place code generation.

We have carried out a case study to experiment and validate the engineering proposition using a compile-time metaprogramming language, a user-interface builder and a general purpose modeling

tool. Overall we were truly impressed by the compositional flexibility which allowed us to safely and easily manipulate and extend the produced interface and application code without suffering from maintenance issues. We believe our work reveals the chances by combining metaprogramming and generative MDE tools, and anticipate more efforts to further exploit this field.

REFERENCES

Actifsource GmbH (2010). *Actifsource Code Generator for Eclipse.* http://www.actifsource.com/_downloads/actifsource_code_generator_for_Eclipse_en.pdf Accessed 29 Oct 2013.

Antkiewicz, M. 2007. Round-trip engineering using framework-specific modeling languages. In Companion to the *22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion* (OOPSLA'07).ACM,927-928. DOI= http://doi.acm.org/10.1145/1297846.1297949.

Badreddin, O. and Lethbridge, T.C. 2013. Model Oriented Programming: Bridging the Code-Model Divide. In *Proceedings of the 5th International Workshop on Modeling in Software Engineering*, IEEE, pp. 69-75, DOI=http://dx.doi.org/10.1109/MiSE.2013.6595299.

Bettini, L. 2013. *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt Publishing.

Calcagno, C., Taha, W., Huang, L. and Leroy, X. 2001. A bytecode-compiled, type-safe, multi-stage language.

Chalabine, M. and Kessler, C. 2007. A Formal Framework for Automated Round-Trip Software Engineering in Static Aspect Weaving and Transformations. *In* Proceedings of the 29th international conference on Software Engineering (ICSE '07). IEEE, 137-146. DOI= http://dx.doi.org/10.1109/ICSE.2007.7.

Desfray, P. 2009. Modelio: Globalizing MDA. In Proc. of the Tools and Consultancy Track of the *Fifth European Conference on Model-Driven Architecture Foundations and Applications* (ECMDA-FA 2009).

Dybvig, R. K. 2009. The Scheme Programming Language (fourth edition). The MIT Press.

Eclipse Foundation. 2008. Eclipse Modeling Framework (EMF). http://www.eclipse.org/modeling/emf/ Accessed 29 Oct 2013.

Fleutot, F. 2007. Metalua Manual. http://metalua.luaforge.net/metalua-manual.html. Accessed 29 Oct 2013.

Hemel, Z., Kats, L. C. L., Groenewegen, D. M. and Visser, E. 2010. *Code generation by model transformation: a case study in transformation modularity*. Software and Systems Modelling, 9(3):375-402. DOI=http://dx.doi.org/10.1007/s10270-009-0136-1.

Lanusse, A., Tanguy, Y., Espinoza, H., Mraidha, C., Gerard, S., Tessier, P., Schnekenburger, R., Dubois, H. and Terrier, F. 2009. Papyrus UML: an open source toolset for MDA. *In Proceedings of the Tools and Consultancy Track of the Fifth European Conference on Model-Driven Architecture Foundations and Applications* (ECMDA-FA 2009).

Lilis, Y. and Savidis, A. 2012. Supporting Compile-Time Debugging and Precise Error Reporting in Meta-Programs. In TOOLS 2012, *International Conference on Technology of Object Oriented Languages and Systems,* Springer LNCS 7304, pp. 155-170. DOI= http://dx.doi.org/10.1007/978-3-642-30561-0_12.

Michotte, B. and Vanderdonckt, J. 2008. GrafiXML, a Multi-target User Interface Builder Based on UsiXML. *In Proceedings of ICAS 2008 4th International Conference on Autonomic and Autonomous Systems*, Gosier, Guadeloupe, IEEE, 15-22. DOI= http://dx.doi.org/10.1109/ICAS.2008.29.

Neverov, G. and Roe, P. 2004. Metaphor: A Multi-Stage, Object-Oriented Programming Language. In *Proceedings of the Third International Conference on Generative Programming and Component Engineering* (GPCE '04), Vancouver, Canada, Springer LNCS 3286, pp. 168-185, DOI=http://dx.doi.org/ 10.1007/978-3-540-30175-2_9.

Obeo. 2006. Acceleo: MDA generator. http://www.acceleo.org/pages/home/en. Accessed 29 Oct 2013.

OMG. 2012. Object Management Group Object Constraint Language (OCL). http://www.omg.org/spec/OCL/ISO/19507/PDF/ Accessed 29 Oct 2013.

Sauer, S. and Engels, G. 2007. Easy model-driven development of multimedia user interfaces with GuiBuilder. *In* Proceedings of the 4th international conference on Universal Access in Human Computer Interaction (UAHCI'07), Beijing, China, Springer LNCS 4554, pp. 537-546, DOI= http://dx.doi.org/10.1007/978-3-540-73279-2_60.

Savidis, A. 2012. Delta Programming Language. Available online from: http://www.ics.forth.gr/hci/files/plang/Delta/Delta.html. Accessed 29 Oct 2013.

Sheard, T. 1999. Using MetaML: *A Staged Programming Language*. In: Advanced Functional Programming. Springer LNCS 1608, pp. 207–239, DOI=http://dx.doi.org/10.1007/10704973_5.

Sheard, T., Benaissa, Z. and Martel, M. 2000. Introduction to Multistage Programming Using MetaML. Pacific Software Research Center, Oregon Graduate Institute, 2nd edition. Available from: http://web.cecs.pdx.edu/~sheard/papers/manual.ps. Accessed 29 Oct 2013.

Sheard, T. and Peyton Jones, S. 2002. Template metaprogramming for Haskell. SIGPLAN Not. 37, 12, 60-75, DOI=http://dx.doi.org/10.1145/636517.636528.

Staiger, S. 2007. Static Analysis of Programs with Graphical User Interface. *In Proceedings of 11th European Conference on Software Maintenance and Reengineering* (CSMR '07). IEEE, pp. 252-264, DOI=http://dx.doi.org/10.1109/CSMR.2007.44.

Steele, G. L. 1990. Common Lisp: *The Language. Digital Press*, second edition.

Taha, W. 2004. A gentle introduction to multi-stage programming. *In Proceedings of Domain-Specific Program Generation*, Springer LNCS 3016, pp. 30-50. DOI=http://dx.doi.org/10.1007/978-3-540-25935-0_3.

Tratt, L. 2005. Compile-time meta-programming in a dynamically typed OO language. *In Proceedings of the 2005 Symposium on Dynamic Languages* (DLS '05). ACM, New York, USA, pp. 49-63, DOI=http://doi.acm.org/10.1145/1146841.1146846.

Vlissides, J. 1996. Generation Gap [software design pattern]. C++ Report, 8(10), pp. 12, 14-18.

Völter, M. and Groher, I. 2007. Handling variability in model transformations and generators. *In Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling* (DSM '07). Available online from: http://www.voelter.de/data/workshops/HandlingVariabilityInModelTransformations.pdf.

Westbrook, E., Ricken, M., Inoue, J., Yao, Y., Abdelatif, T. and Taha, W. 2010. Mint: Java multi-stage programming using weak separability. *In Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '10 ACM, New York, 400-411. DOI=http://doi.acm.org/10.1145/1806596.1806642.

wxFormBuilder. 2006. wxFormBuilder - A RAD tool for wx GUI design. http://sourceforge.net/projects/wxformbuilder/ Accessed 29 Oct 2013.

Zschaler, S. and Rashid, A. 2011. Towards modular code generators using symmetric language-aware aspects. In Proceedings of the *1st International Workshop on Free Composition* (FREECO '11). ACM, New York, NY, USA, Article 6, 5 pages. DOI=http://doi.acm.org/10.1145/2068776.2068782.