IMPROVED MODEL-DRIVEN ENGINEERING WITH STAGED CODE GENERATORS

Yannis Valsamakis

Thesis submitted in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes, Heraklion, GR-70013, Greece

Thesis Advisor: Prof. *Anthony Savidis*

University Of Crete
Computer Science Department

IMPROVED MODEL-DRIVEN ENGINEERING WITH STAGED CODE GENERATORS

Thesis submitted by
Yannis Valsamakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____

Yannis Valsamakis, Department Of Computer Science

Committee approvals: _____

Anthony Savidis
Professor, Thesis Supervisor

_____

Yannis Tzitzikas
Assistant Professor, Committee Member

_____

Irini Fundulaki
Principal Researcher of ICS-FORTH, Committee Member

Department approval: _____

Angelos Bilas
Professor, Director of Graduate Studies

Heraklion, October 2013

Abstract

The amount of software systems' source code today practically explodes. Commercial software systems such as games consist of hundreds of thousands lines of code. The main challenge of developing such systems is connected with maintainability and extensibility issues. The software engineering of such systems focuses on the design level, where the use of advanced programming techniques is mandatory.

Model-Driven Engineering (MDE) is an advanced programming technique based on model reuse and evolution. In general, MDE involves tools, models, processes, methods and algorithms addressing the demanding problem of (semi-)automated generation of source code. On the one hand the involved tools improve the deployment of MDE; on the other hand they cause several problems. One of the most challenging problems is the maintenance issue inherent in model-driven code generators. The problem appears in the development life cycle. In particular, the auto-generated source code is altered and supplemented manually by developers to complete the developing project. The manually-written source code is overwritten by the code re-generation caused by the generative MDE tool. Additionally, while MDE is a widely used software engineering approach it is typically practiced separately from the rest of the development process that takes place within an Integrated Development Environment (IDE). Therefore, there are numerous MDE tools included as plugins of some IDEs, however a large number of them cannot be properly incorporated.

In this thesis, we propose an alternative approach for MDE, using an advanced programming feature; metaprogramming, which is supported by several languages. Our approach is based on the following principles: (i) the MDE tool is invoked as part of the metaprogram evaluation; (ii) instead of generating code, the MDE tool generates source fragments as abstract syntax trees (ASTs); (iii) the generated source fragments are directly inserted into the main program source through generator

macros of the metaprogram; and (iv) the resulting program that incorporates both model code and custom application code can be normally compiled to produce the final application.

We have carried out the proposed approach in the Delta programming language, which supports compile-time metaprogramming. Finally, we have deployed enough case studies to test the validity and the effectiveness of our approach.

ΒΕΛΤΙΩΜΕΝΗ ΟΔΗΓΟΥΜΕΝΗ ΑΠΟ ΜΟΝΤΕΛΑ ΑΝΑΠΤΥΞΗ ΛΟΓΙΣΜΙΚΟΥ ΜΕ ΓΕΝΝΗΤΡΙΕΣ ΚΩΔΙΚΑ ΠΟΥ ΕΦΑΡΜΟΖΟΝΤΑΙ ΠΡΙΝ ΤΗ ΜΕΤΑΓΛΩΤΤΙΣΗ

Περίληψη

Το μέγεθος του πηγαίου κώδικα των συστημάτων λογισμικού σήμερα αυξάνεται εκθετικά. Τα εμπορικά συστήματα λογισμικού όπως τα παιχνίδια αποτελούνται από εκατοντάδες χιλιάδες γραμμές κώδικα. Το κυρίως πρόβλημα της ανάπτυξης ενός συστήματος τέτοιας κλίμακας συνδέεται με θέματα συντήρησης και επέκτασης. Η τεχνολογία ανάπτυξης τέτοιων συστημάτων επικεντρώνεται κυρίως στο σχεδιαστικό επίπεδο, όπου η χρήση προηγμένων προγραμματιστικών τεχνικών κρίνεται απαραίτητη.

Η ανάπτυξη λογισμικού οδηγούμενη από μοντέλα (Model-Driven Engineering, MDE) είναι μία προηγμένη προγραμματιστική τεχνική η οποία βασίζεται στη δημιουργία, επαναχρησιμοποίηση και εξέλιξη μοντέλων. Γενικά, το MDE επικαλείται εργαλεία, μοντέλα, διεργασίες, μεθόδους και αλγόριθμους που αντιμετωπίζουν το απαιτητικό πρόβλημα της (ημί-) αυτόματης παραγωγής πηγαίου κώδικα. Από τη μια πλευρά η χρησιμοποίηση εργαλείων βελτιώνει την εφαρμογή του MDE, από την άλλη όμως προκαλούνται αρκετά προβλήματα. Ένα από τα πιο σοβαρά προβλήματα αφορά θέματα συντήρησης που βρίσκονται εγγενώς στα εργαλεία παραγωγής κώδικα που βασίζονται σε μοντέλα. Το πρόβλημα αυτό εμφανίζεται κατά την διάρκεια του κύκλου ανάπτυξης λογισμικού. Συγκεκριμένα, ο αυτόματα παραγόμενος πηγαίος κώδικας τροποποιείται και συμπληρώνεται κατάλληλα από τους προγραμματιστές ώστε να ολοκληρωθεί το υπό ανάπτυξη έργο. Ο πηγαίος κώδικας που προστίθεται με το χέρι χάνεται όταν ξαναδημιουργηθεί ο αυτόματα παραγόμενος πηγαίος κώδικας από ένα μοντέλο. Παρόλο που το MDE χρησιμοποιείται ευρέως ως μεθοδολογία κατασκευής λογισμικού, συνήθως είναι απομονωμένο από την υπόλοιπη διαδικασία που λαμβάνει χώρα σε ολοκληρωμένα περιβάλλοντα ανάπτυξης (IDE). Παρότι υπάρχουν MDE εργαλεία που διατίθενται ως

επεκτάσεις σε κάποια IDE, ωστόσο ένας μεγάλος αριθμός από αυτά δεν ενσωματώνονται επαρκώς.

Σε αυτή την εργασία, προτείνουμε μια εναλλακτική προσέγγιση για MDE, χρησιμοποιώντας ένα προηγμένο χαρακτηριστικό, τον μετα-προγραμματισμό, ο οποίος υποστηρίζεται από αρκετές γλώσσες προγραμματισμού. Η προσέγγιση μας είναι βασισμένη στις ακόλουθες αρχές: (i) το MDE εργαλείο επικαλείται σαν μέρος της αποτίμησης του μετα-προγράμματος, (ii) αντί να παραχθεί πηγαίος κώδικας, το MDE εργαλείο δημιουργεί τμήματα κώδικα στην μορφή αφηρημένων συντακτικών δέντρων (AST), (iii) τα δημιουργημένα τμήματα κώδικα εισάγονται άμεσα στο κυρίως πηγαίο πρόγραμμα μέσω μακροεντολών του μετα-προγράμματος, και (iv) το πρόγραμμα που προκύπτει ενσωματώνει τον αυτόματα παραγόμενο κώδικα με τον επιπλέον κώδικα της εφαρμογής και μεταγλωττίζεται κανονικά ώστε να παραχθεί η τελική εφαρμογή.

Έχουμε εφαρμόσει την προτεινόμενη προσέγγιση στην γλώσσα προγραμματισμού Delta, η οποία υποστηρίζει μετα-προγραμματισμό κατά τη μεταγλώττιση (compile-time metaprogramming). Τέλος, έχουμε αναπτύξει αρκετά σενάρια χρήσης ώστε να ελέγξουμε την εγκυρότητα και την αποτελεσματικότητα της προσέγγισής μας.

Acknowledgements

First of all, I would like to thank my supervisor, professor of the University of Crete, Anthony Savidis, initially for trusting me and then for his continuous support and his valuable advice. I would also like to thank Yannis Lilis for his excellent cooperation and for his continuous support. I am also grateful to the professors Yannis Tzitzikas and Irini Fundulaki for participating in the supervisory committee. I would also like to thank the Computer Science Department of Greece for offering a high level of academic education and the HCI Laboratory of ICS-FORTH for providing a high-level research environment.

I would also like to thank my friends for supporting me all through this period. Finally, most of all, I would like to thank my parents Flora and Nikos. I am grateful for all their love and support. Without them, I would not be the person I am today.

Στην οικογένεια μου

Contents

Abstract 4

Περίληψη 6

Acknowledgements 8

Contents 10

List of Figures 13

List of Tables 18

1. Introduction 20

1.1 Model-Driven Engineering 20

1.2 Multistage Languages 23

1.3 Problem Definition 27

1.4 Primary Contributions 29

1.5 Thesis structure 30

2. Related Work 31

2.1. General Purpose MDE tools 31

2.2. Specific Mission MDE tools 40

3. Improved Process 46

3.1 Tool Chain 47

3.1.1 Invocation 48

3.1.2 Deployment 50

3.2 Producing ASTs 52

3.3 Transforming ASTs 53

3.3.1 Batches - Separate Metaprograms 53

3.3.2 Stages - Embedded Metaprograms 55

3.3.3 Combining Batches and Stages 55

3.4 Unparsing ASTs 56

4. Case Studies 58
4.1 User Interface Builder 58
4.1.1 Applying our approach for UIs 59
4.1.2 Developing User Interfaces 66
4.2 Class Builder 71
4.2.1 Applying our approach for Class Hierarchy 71
4.2.2 Developing Applications 73
4.3 Automatic User Interfaces 77
4.3.1 Defining an alternative UI model 78
4.3.2 The Auto-generation UIAPI engine 82
4.3.3 User-Interface Design Issues 84
4.3.4 Developing UI for a Library 86
4.4 Combined Deployment 89
4.4.1 Developing a paint application 90
4.4.2 Developing a library application 92
5. Discussion 94
5.1 Maintenance 94
5.1.1 Addressing maintenance issues so far 95
5.1.2 How our approach solves maintenance 97
5.2 Tradeoffs of our approach 100
5.3 Applicability of our approach 101
6. Conclusions and Future Work 103
Bibliography 106

List of Figures

Figure 1. The core idea of Model-Driven Engineering..... 21
Figure 2. *Top*: high-level overview of model-driven processes outlining the general tool roles and respective input / output links; *Bottom*: Architecture of generative model-driven tools: (1) interactive model editing; (2) code generation from models; and (3) tags inserted in the generated source code to carry model information and enable model reconstruction. 22
Figure 3. Separation between MDE deployment and the remaining development process.. 23
Figure 4 Evaluation of generative macros with an extra stage. 24
Figure 5 Example of an abstract syntax tree for three statements using the wx widgets library: (i) *left*: creating a frame widget; (ii) *middle*: setting its size; and (iii) *right*: creating a text widget. 25
Figure 6 Common growth of application code around the originally generated code; future custom extensions and updates eventually lead to bidirectional dependencies. 27
Figure 7 The primary maintenance issues in the deployment of generative model-driven tools either individually (left) or collectively (right). 28
Figure 8. Using the EMF tool in Eclipse; Area 1 is the Palette toolbar of the Model constructs; Area 2 is the *"action"* area of Models construction; Area 3 is the view/edit the data constructs of the Models; Area 4 is the project explorer of Eclipse Platform. 32
Figure 9. Using the *Actifsource* tool in Eclipse Platform; Area 1 is the *"action"* area of model construction; Area 2 is the Palette toolbar of the Model constructs; in Area 3 you can view/edit the data of constructs of models; Area 4 is the navigation of project tool. 33
Figure 10. Using the *Umple* tool online version; Area 1 is the *"action"* area of model construction; Area 2 is the toolbar of *Umple*; in Area 3 you can view/edit the code that will be generated. 34
Figure 11. Using the Papyrus; Area 1 is the *"action"* area of model construction; Area 2 is the Palette toolbar of the Model constructs; in Area 3 you can view/edit the data of constructs of models; in Area 4,5 you can see in two different ways the outline of the model; Area 6 is the navigation of project tool. 36
Figure 12. Using the *Modelio* tool; Area 1 is the *"action"* area of model construction; Area 2 is the toolbar of the Model constructs. 37

Figure 13. Using the *Altova UModel*; Area 1 is the *"action"* area of model construction; Area 2 is the toolbar of the Model constructs; in Area 3 you can view/edit the data of constructs of models; in Area 4 you can see the Diagram model tree view. 38
Figure 14. Using the *Enterprise Architect*; Area 1 is the *"action"* area of model construction; Area 2 is the toolbar of the Model constructs; in Area 3 is the project navigation of the tool. 39
Figure 15. Using the *Apollo* tool; Area 1 is the *"action"* area of model construction; Area 2 is the Palette toolbar of the Model constructs; in Area 3 you can see the outline of Model's diagram; in Area 4 is the project explorer of the *Apollo* tool..... 40
Figure 16. Using the wxFormBuilder; Area 1 is the *editor* of the UI model construction; Area 2 is the widgets toolbar of the UI Model constructs; Area 3 is the view/edit the widgets' properties-events; in Area 4 is the tree view of the constructed UI Model 41
Figure 17. Using the *GrafiXML*; Area 1 is the *editor* of the UI model construction; Area 2 is the toolbar of the UI Model constructs; Area 3 is the view/edit the properties-events; in Area 4 is the project explorer of the GrafiXML tool. 42
Figure 18. Using the *Glade*; Area 1 is the *editor* of the UI model construction; Area 2 is the widgets toolbar of the UI Model constructs; Area 3 is the view/edit the widgets' properties; in Area 4 is the tree view of the constructed UI Model. 42
Figure 19. Using the wxGlade; Area 1 is the *editor* of the UI model construction; Area 2 is the widgets toolbar of the UI Model constructs; Area 3 is the view/edit the widgets' properties; in Area 4 is the tree view of the constructed UI Model. 43
Figure 20. Using the wxDesigner; Area 1 is the *editor* of the UI model construction; Area 2 is the widgets toolbar of the UI Model constructs; Area 3 is a view/edit dialog of widgets (opening when double click in the widget in the editor); in Area 4 is the tree view of the constructed UI Model..... 44
Figure 21. Using the *Blend*; Area 1 is the *editor* of the UI model construction; Area 2 is the toolbar of the UI Model constructs; Area 3 is a view/edit widgets properties; in Area 4 is the project explorer of the *Blend* tool. 45
Figure 22. Encapsulating the model-driven process directly in the application source through staged metaprograms. *Step 1*: Staged code execution macros invoke the MDE tool that creates the model and converts its corresponding code as ASTs. *Step 2*: Staged code generator macros take the ASTs as input and insert the model-driven code into the source along with custom application code. *Step 3*: The transformed source is normally translated or evaluated to produce the final binary of the entire application. 46

Figure 23. Invocation of MDE tools in the beginning of build process through staged code. 49
Figure 24. Deployment of MDE tools in the development process 51
Figure 25. Running meta-programs which load the binary files of AST; transform and save it back to the disk. 54
Figure 26. In the left part, we can see a form of our approach source code; in the right part the source code of a classic model-driven process. 57
Figure 27. Deployment of approach focusing on User Interface builder. 59
Figure 28. Code generation; in label 1 the application for a pure dialog code with embedded staged code is outlined; in label 2 there is the result of the build process (i.e. the generated code which is read-only); in label 3 the pure dialog is depicted. 60
Figure 29. Overview of the compile-time MDE deployment through staged metaprogramming. Actions performed during the metaprogram execution (top right) and their corresponding source code lines (bottom) are shown with matching numbers 61
Figure 30. Cut UI parts of Calculator in order to transform scientific calculator in a simple calculator 63
Figure 31. Crop the auto-generated frame from Shapes' toolbar User-Interface. 63
Figure 32. Merge two independent UI code; Calculator and Calendar UIs in one UI application. 64
Figure 33. Create & Insert Shapes' User-Interface toolbar in the Paint's application User-interface. 65
Figure 34. Editing the tab's text "Home" of the Paint's application User-Interface. 65
Figure 35. Two example scenarios (middle, right) of user-interface source code composition relying on AST manipulation on top of the original GUI authored with the interface builder (left); updates on the scenarios are automated and are directly remapped on top of the original GUI by simple performing recompilation. 67
Figure 36. The GUI parent object typically required 67
Figure 37. Meta-code to load, manipulate (four labeled steps) and inline the source code for the modified calculator. 68
Figure 38. Examples of the generated interfaces: Left: Original application GUI authored by the interface builder; Middle: Custom toolbar authored as a separate interface; Right: Composing the two previous interfaces through AST manipulation. 69
Figure 39. The dialog open at compile-time to handle the models of development 70
Figure 40. Deployment of approach focusing on Class builder 71

Figure 41. The internal custom model editor launched during compilation case study of Geometry..... 72
Figure 42. The internal editor code as an inherent part of the staged metaprogram..... 73
Figure 43. Top-left: Ecore model of the target class hierarchy; Top-right: Code structure (AST) generated by the model; Bottom: Deployment code for loading and converting the model to AST, performing manual updates through AST editing and inlining the final AST code. The initial value of the meta-variable ast corresponds to the code structure shown at top-right.74
Figure 44. Supporting quick access to all class hierarchy entities through AST decoration. The AST shown corresponds to the generated hierarchy of Figure 43, while the highlighted path `ast.Geometry.Circle.area.body` was used to insert custom method functionality. 75
Figure 45. Meta-code to load, manipulate and inline the source code for the library..... 77
Figure 46. List's User Interface description; at the top there is the description when dataFlowType is "In" or "InOut"; at the bottom there is the description when dataFlowType is "Out"..... 81
Figure 47. String's User-Interface description; on the left side is the UI when *dataFlowType* is "In" or "InOut"; on the right side is the UI when dataFlowType is "Out"..... 82
Figure 48. The auto-generation UI engine architecture..... 83
Figure 49. Default embedded metacode using the auto-generation tool we developed 85
Figure 50. User-Interface produced by auto-generation UI engine..... 87
Figure 51. Manipulating the auto-generated User-Interface through ASTs transformations. 87
Figure 52. Meta-code to include the specification(model UI), the UIAPI engine and the library of manipulating UI for AST's operators in label 1; Meta-code to call the auto UIAPI engine in label 2; Meta-code to transform the auto-generated AST's GUI in labels 3,4,5 and inline the ASTs in order to generate the Library's application source code in label 6..... 88
Figure 53. Deployment of approach focusing on combined deployment; use more than one MDE tools..... 89
Figure 54. 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 right of Figure 38..... 91
Figure 55. Left: Ecore model of the target class hierarchy; Right: Code structure (AST) generated by the model..... 93
Figure 56. Using EMF tool to design and implement class Person..... 95

Figure 57. 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, fill, edit source code in the form of ASTs and generate a transient code version that will be integrated along with the custom application. 98
Figure 58. Developing a Person example in our approach and the result of the generated code in label 4. 99
Figure 59. Applying the generative MDE process with runtime 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. 102

List of Tables

Table 1. Comparing the approaches which deal with maintenance issues 100

Chapter 1

Introduction

1.1 Model-Driven Engineering

Model-Driven Engineering (MDE) [1] is an approach in software development which focuses on creating and exploiting domain models. These models include abstract representations of the knowledge and activities that govern a particular application domain, rather than on the computing or algorithmic concepts. The general philosophy of MDE rents its roots to Model-Driven-Architecture (MDA) of the Object Management Group [2], emphasizing accelerated (rapid) application development together with model-oriented reuse and evolution.

The core idea of Model-Driven Engineering is depicted in Figure 1. Using the MDE tools Platform Independent (PI) Models are constructed. Then, it is possible to capitalize on PI models, use them to automatically derive Platform-Specific (PS) models through transformation engines and ultimately utilize code generators to automatically produce the source code corresponding to the modeled entities. The auto-generated source code can then be extended or linked with custom application code to deliver the final application. During the development life cycle PI Models can be edited. In this case, the PS Models have to be re-constructed with the use of transformation engines and the

auto-generated source code has to be re-generated in order to update the changes. Moreover, PS Models can be edited in the development life cycle. In this case, the auto-generated source code has to be regenerated in order to transfer the changes from model to source. In addition, the PI models can be kept updated by appropriate Transformation engines during the development life cycle. In general, it is necessary to keep all levels of model abstraction updated so as to have the ability to extend any of them during development life.

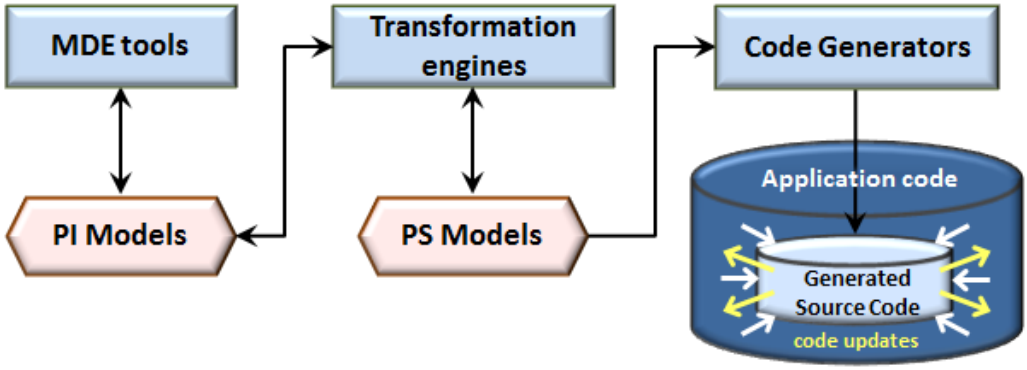


Figure 1. The core idea of Model-Driven Engineering

Additionally, apart from model to model and model to text transformations there are text to model transformations. In particular, during the development life cycle the auto-generated source code is extended. The model which generates it is not updated according to the extensions of the source code. So, appropriate tools are used to create models from the source code (i.e. Model-Driven reverse Engineering). A lot of difficulties appear in this process and it is not always feasible to succeed. There is an example for general purpose MDE tools in literature like *Papyrus* [22] and *Modelio* [26] which support full development life cycle and are described in the next chapter.

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 derived, usually with various intermediate transitions from the abstract to the final implementation domain. This discipline is outlined under Figure 2, showing the specialization from abstractions to instances as a transformation process.

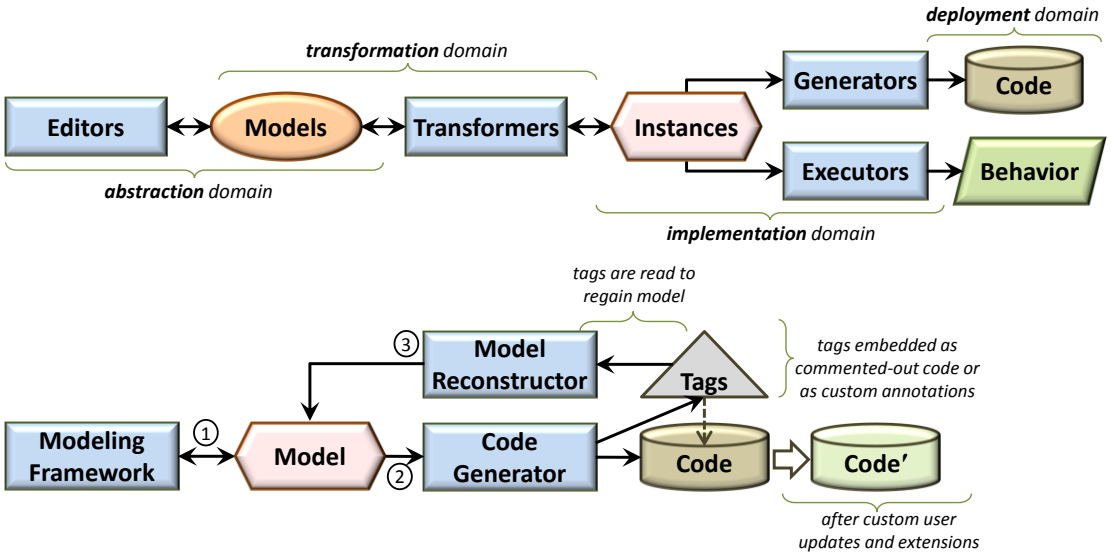


Figure 2. *Top*: high-level overview of model-driven processes outlining the general tool roles and respective input / output links; *Bottom*: Architecture of generative model-driven tools: (1) interactive model editing; (2) code generation from models; and (3) tags inserted in the generated source code to carry model information and enable model reconstruction.

Additionally, there are two categories of model driven tools distinguished by the way their outcomes can be deployed: generative tools, producing source code, and executors, offering custom runtimes which instantiate the behavior of their input models. On the one hand, the former concerns tools supporting a modeling-all-the-way discipline, with emphasis shifted in eliminating the need for manually written source code. On the other hand, the latter relates to tools which automate the engineering of various demanding system features, however, still relying on hand-written source code to complete a fully-fledged system. We consider both universes to be equally valuable and useful in the model-driven tool arena, however in this thesis we focus on generative model-driven tools, and improve them so as to address a maintenance issue they cause.

Furthermore, the auto-generated source code can then be extended or linked with custom application code to deliver the final application. While MDE is a widely used software engineering approach it is typically practiced separately from the rest of the development process (Figure 3). An MDE tool is used to create a model and generate its corresponding source code while that code is then incorporated into an integrated development environment (IDE) for further processing and linking with the remaining application code. In this sense, MDE requires third party tools that cannot always be properly integrated in the deployed IDE.

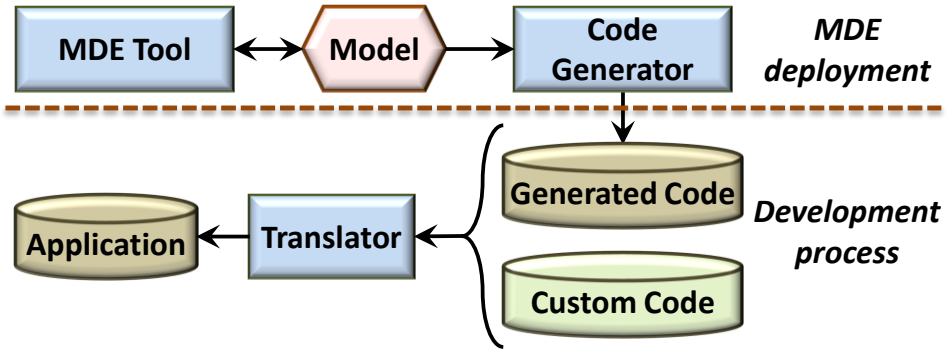


Figure 3. Separation between MDE deployment and the remaining development process.

So, we mainly focus on the maintenance issue we address in our work and also dealing with bringing the MDE deployment as close as possible to the actual application development.

1.2 Multistage Languages

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 [3]. Alternatively, if offered during runtime – usually built on top of 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 [4] [5]. In our work we use Delta [6], a recent publicly available dynamic object-object language along with its compile-time metaprogramming extension [7]. Popular meta-languages include Lisp [8], Scheme [9], MacroML [10], MetaOCaml [11], MetaLua [12] and Converge [13].

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.

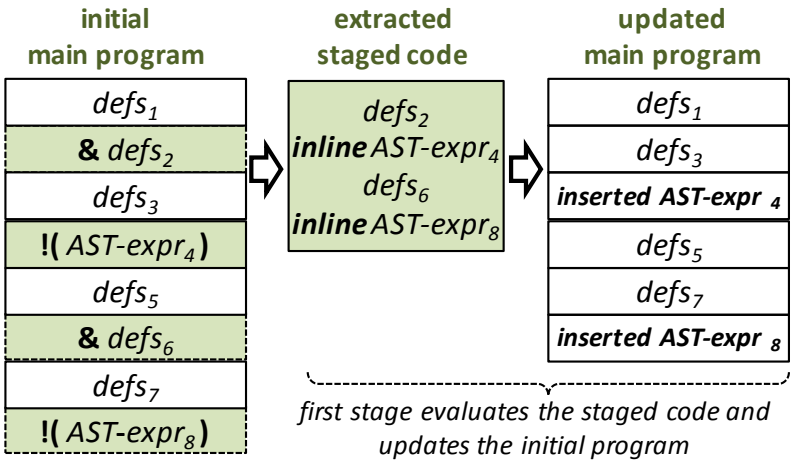


Figure 4 Evaluation of generative macros with an extra stage.

As shown in Figure 4, during the first stage the compiler: (i) collects all scattered meta-code into a single metaprogram; (ii) evaluates the program while internally 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 actually produced. For example, consider the following Delta code.

```
using wx;
&ast = ui::load_ast ("<some ast path>");
!(ast); ← code generation (inline) directive
```

The first line is normal code, a typical directive to import the *wxWidgets* GUI library. But the next two lines are meta-code, distinguished by **&** and **!** prefixes. The second line loads an AST from a file, assume the loaded AST to be the one of Figure 5. 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 wx;
frm = wx::frame_construct(nil, "ID_ANY", "frame");
frm.setsize(wx::size_construct(450, 304));
txt = wx::textctrl_construct(frm, "text");
```

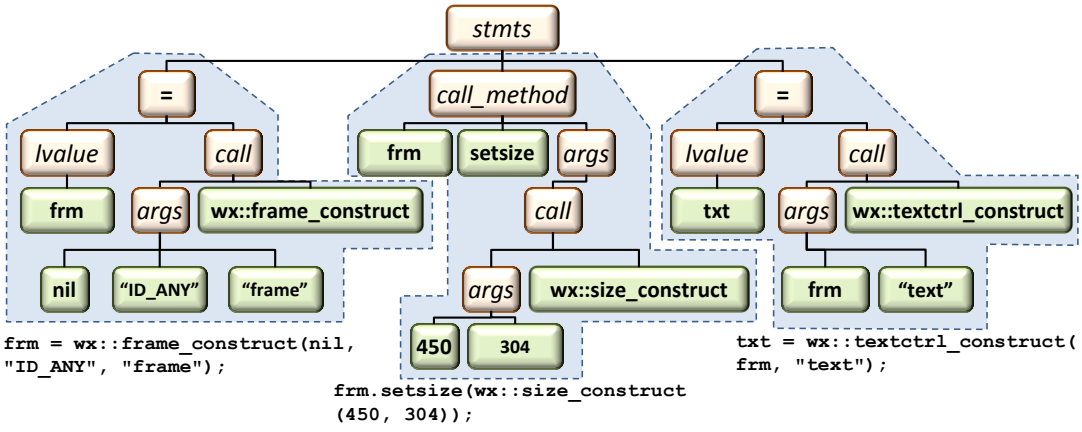


Figure 5 Example of an abstract syntax tree for three statements using the wx widgets library: (i) *left*: creating a frame widget; (ii) *middle*: setting its size; and (iii) *right*: creating a text widget.

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.

In this example, the generated code implicitly depends on the manually written code requiring that it imports the wxWidgets library to allow its usage in the generated code. In a more elaborate case, the code to be inserted will be associated with metadata specifying any such dependencies and thus allowing them to be generated as well. For instance, the first line of the above example could have been generated by the following code.

```
!(load_deps());←loads AST of 'using wx;'
```

Such metadata can provide a more structured usage of the loaded ASTs enabling establishing standardized interfaces between the generated code segments and the rest of the code. For example consider the following code:

```
&data=load_metadata("<path>");
!(load_ast(data.dependencies));
 …other normal program dependencies here…
!(load_ast(data.definitions));
 …other normal program definitions here…
!(load_ast(data.main_code));
 …other normal program program code here…
function f (!(load_ast(data.f_args)))
{
      !(load_ast(data.f_body));
}
```

The loaded metadata are expected to identify the ASTs for any dependencies, definitions and main code so that they are loaded and incorporated in the final code. This also provides a clear interface for manually inserted code that may depend on generated code segments and thus should be placed after the corresponding generative directive. Finally, the granularity of the generated code and the allowed generative directive locations are not limited to top-level code segments, but includes multiple forms and locations. For instance, the above code loads an AST containing a list of statements in order to generate the body of a function. Overall, the AST representation and the code generation scheme offer considerable flexibility, allowing programmers select how fine-grained or coarse grained the source code fragments should be based on the deployment scenario.

The previous examples show 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 generate the above code but replacing the last assignment with a print statement. To achieve this, we would have to obtain and manipulate the children of the root stmts node:

```
&ast = ui::load_ast ("<some ast path>");
&children = ast.get_children(); ← get children from stmts
&children.removeLast(); ← drop last statement
&children.insertLast(<<std::print("<Hello");>>); ← replace it
!(ast); ← generate the 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 merely the character string '1+2'.

1.3 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 the generated code from the rest of the application code, one can never exclude that the possibility that interdependencies or custom updates may appear.

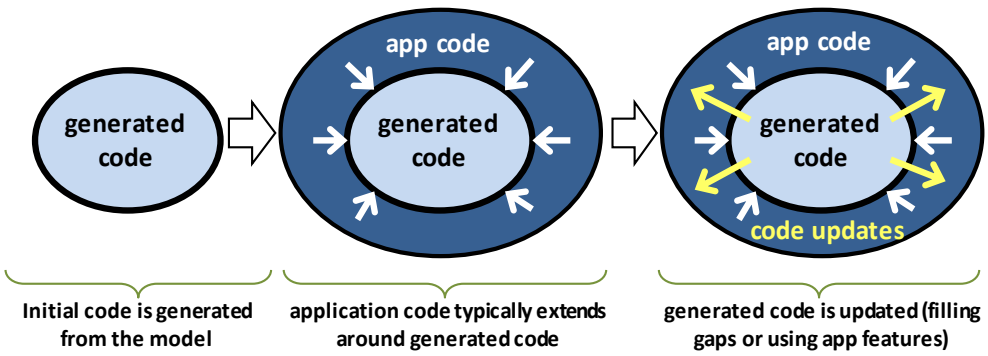


Figure 6 Common growth of application code around the originally generated code; future custom extensions and updates eventually lead to bidirectional dependencies.

The typical lifecycle of the generated code is outlined under Figure 6. 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 invoking, application-specific event handling, linkage to third-party libraries that are not known to the model-driven 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 7. 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.

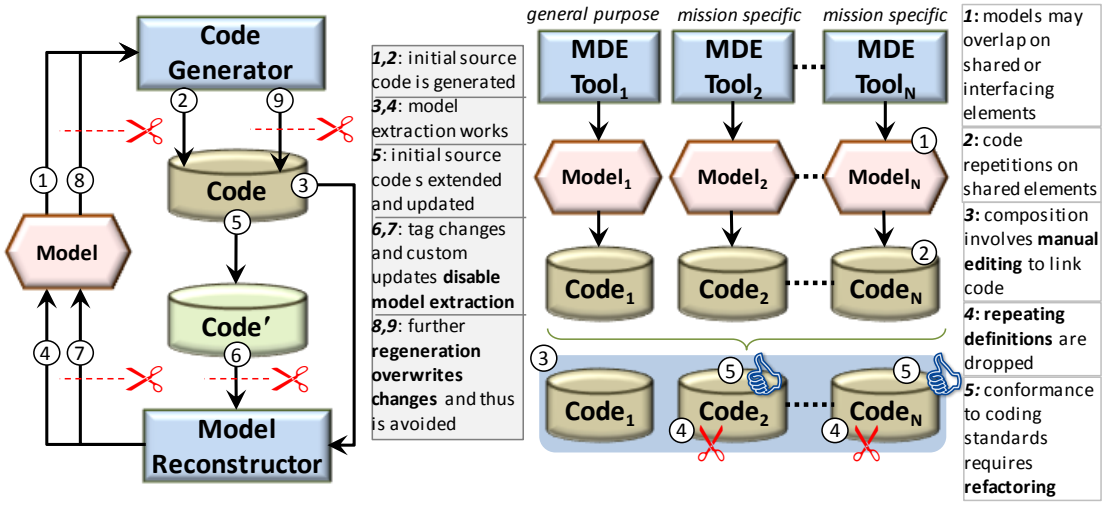


Figure 7 The primary maintenance issues in the deployment of generative model-driven tools either individually (left) or collectively (right).

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).

1.4 Primary Contributions

Our main contribution is an inversed responsibility model for generator MDE tools where: (i) the code for implementing model entities becomes available in the form of ASTs; and (ii) the actual code generation is applied *on-demand* and *in-place* through metaprograms (macros) that are included in the implementation of the main program and are evaluated at compile-time (i.e. during the build process). This approach, not only addresses the maintenance issues of traditional generators, but also sets code manipulation as a first-class concept in the model-driven engineering and reveals the value of using a metaprogramming language in this context.

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.

Additionally, we explore the option of adopting metaprogramming practices to allow specifying the deployment of an MDE tool directly in the program source. Essentially, we propose launching the MDE tool and generating the model code as

part of the metaprogram. Then the generated code along with the manually edited source code can be normally compiled to produce the final application.

1.5 Thesis structure

The rest of this work is organized as follows; In Chapter 2, we review most advanced and popular general-purpose related MDE tools. Chapter 3 follows, which is the main core of this thesis, where our proposal for an improved model-driven approach is described. It begins by outlining the steps of the improved model-driven engineering and then analyzes the 'key' steps of the proposed approach in each of the subsections. Chapter 4 gives a description of the Case Studies, we have carried out in order to test the proposed MDE approach of our work and assess the expressive power and its engineering validity. Chapter 5 concludes the work and identifies issues for further research work.

This work has resulted in the publication of the following paper:

Self Model-Driven Engineering Through Metaprograms, Yannis Lilis, Anthony Savidis and Yannis Valsamakis, PCI 2013, September 19 - 21 2013, Thessaloniki, Greece

Through the following link you can download, view and use the deployment of our approach using specific MDE tools and a specific language which supports metaprogramming and their Case Studies:

https://app.box.com/mdewithstagedcodegenerators

Chapter 2

Related Work

In this chapter, we review the most advanced and popular MDE tools. We focus on each tool which addresses (or not) issues relevant to maintenance which we solve in this thesis through our approach in order to improve the MDE process. We begin by reviewing general-purpose MDE tools and then we review specific mission MDE tools.

2.1. General Purpose MDE tools

Acceleo

Acceleo [16] is an open-source code generator from the Eclipse Foundation, 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 [17] template-like definition for expressing custom generators. Acceleo supports incremental generation allowing developers to regenerate target files without losing any modifications. This is achieved by the use of explicit `[protected] … [/protected]` constructs that are translated into

tagged comments and mark a code region that will not be overwritten during regeneration. Nevertheless, any developer intervention on such generated tags may break regeneration. Furthermore, the placing of 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. Using our approach, any code updates are performed directly in the source file while the generated model code, available in a read-only form, is explicitly deployed on-demand and in-place through metaprogramming.

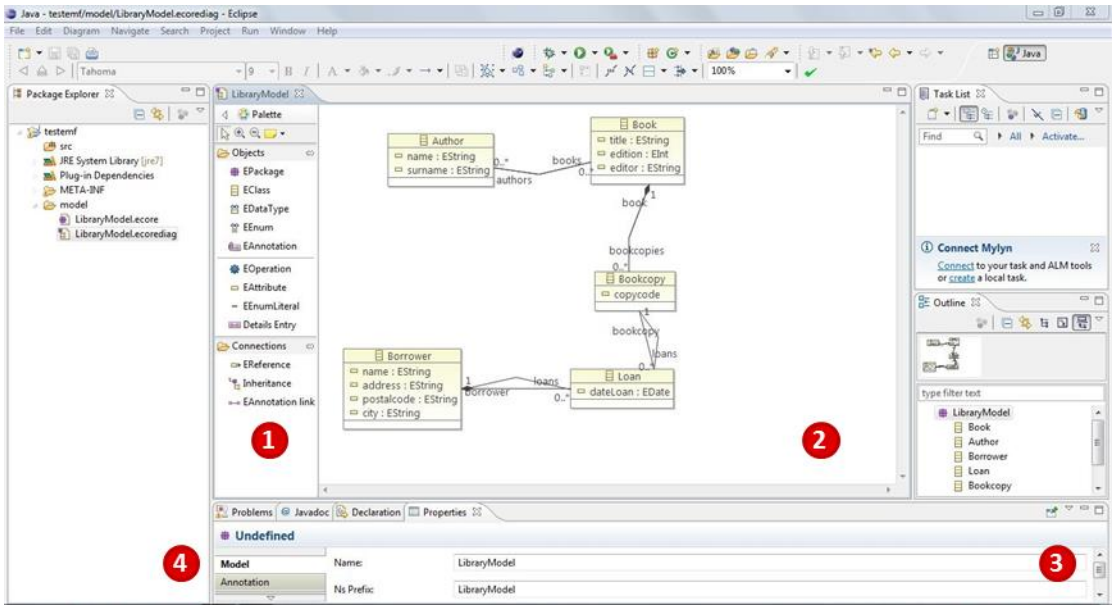


Figure 8. Using the EMF tool in Eclipse; Area 1 is the Palette toolbar of the Model constructs; Area 2 is the *"action"* area of Models construction; Area 3 is the view/edit the data constructs of the Models; Area 4 is the project explorer of Eclipse Platform.

EMF tool

The Eclipse Modeling Framework (EMF) [14] is a MDE tool plugin of Eclipse [15]. The *EMF* project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. The model itself is described using the *Ecore* meta-model, while the code generation targets Java and utilizes the annotation `@generated` 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 removed 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. In the below figure is depicted the *EMF tool* during construction of a Library model.

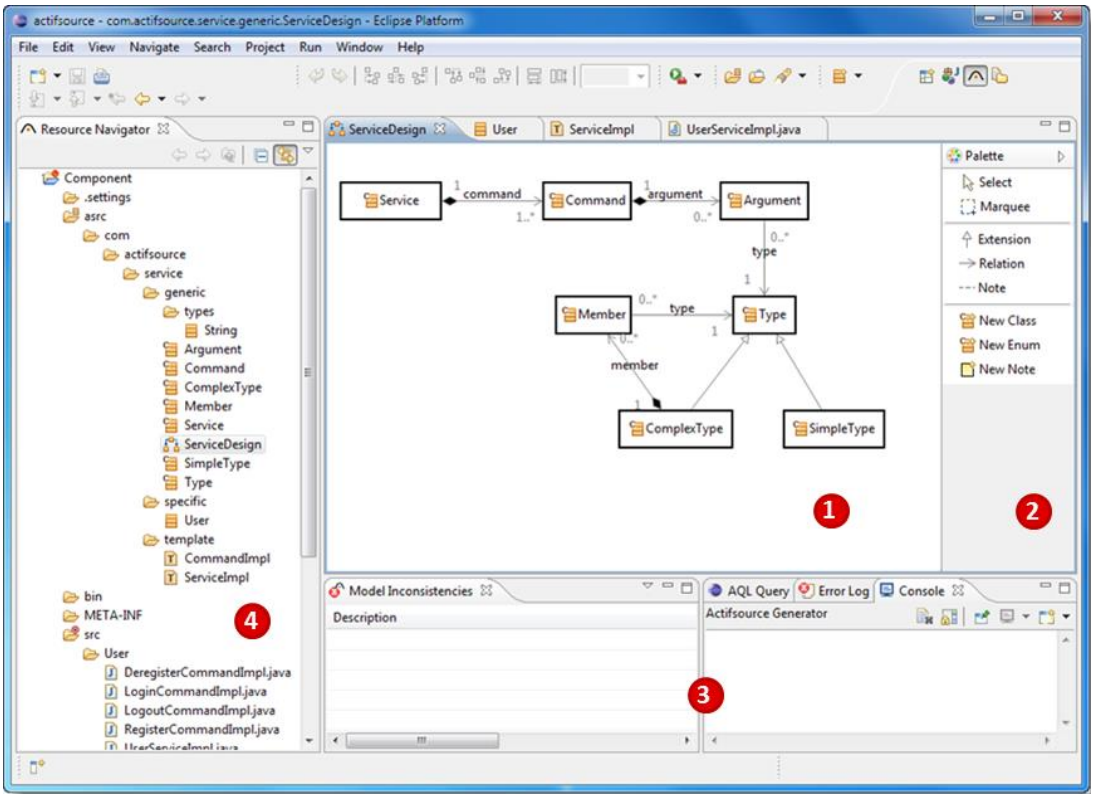


Figure 9. Using the *Actifsource* tool in Eclipse Platform; Area 1 is the *"action"* area of model construction; Area 2 is the Palette toolbar of the Model constructs; in Area 3 you can view/edit the data of constructs of models; Area 4 is the navigation of project tool.

Actifsource

Actifsource [18] 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, however, any developer intervention on these tags will cause maintenance issues when the code is regenerated. In the Figure 9 is depicted the *Actifsource* during construction of a model.

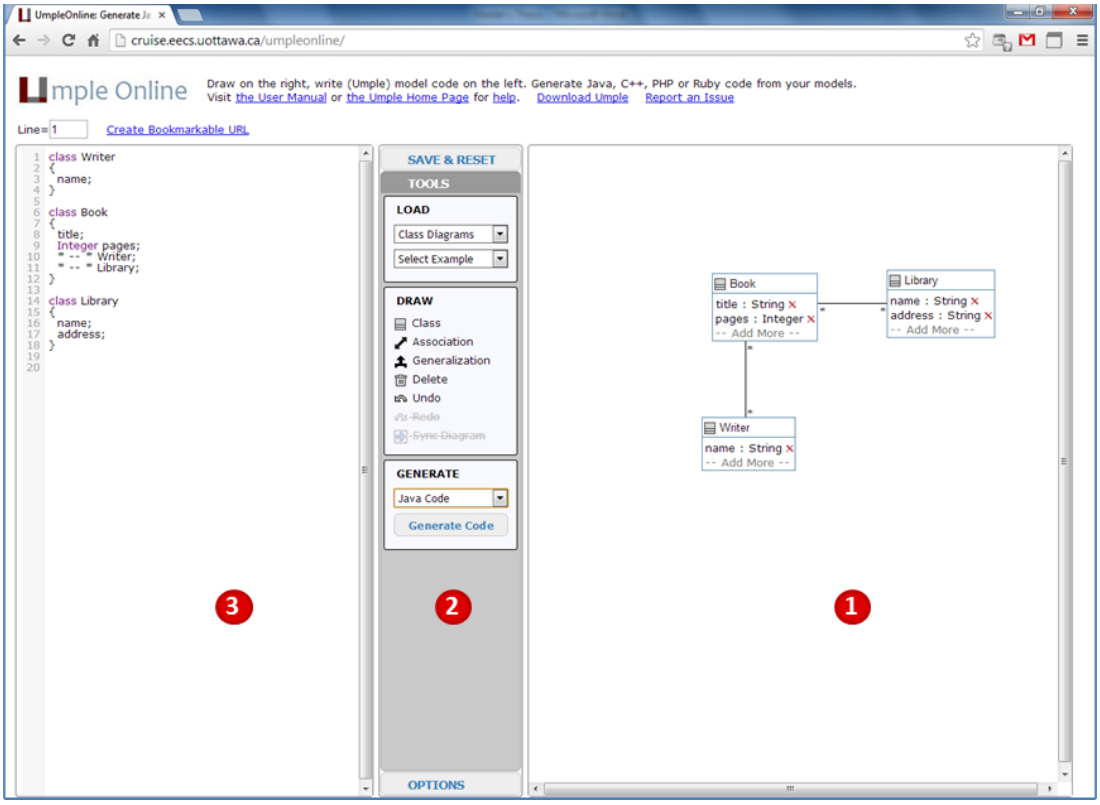


Figure 10. Using the *Umple* tool online version; Area 1 is the *"action"* area of model construction; Area 2 is the toolbar of *Umple*; in Area 3 you can view/edit the code that will be generated.

Umple

Umple [19] is a modeling tool that tries to reduce the distance between model and code by introducing UML abstractions directly into a high-level programming language code. This way, models become just another abstract view of the code and the need for extracting the model from the code is eliminated as everything in the model is represented directly in the 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 [20] [21]. 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. In the Figure 10 is depicted the *Umple* online version during construction of a model.

Papyrus

Papyrus [22] is an open source UML 2 tool based on Eclipse platform and licensed under the EPL. It can either be used as a standalone tool or as an Eclipse plug-in. Papyrus is a model-driven tool offering code generation for a variety of languages. It supports the full MDE development life cycle allowing both model-to-source and source-to-model transformations. In order to provide the latter, it parses 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 full MDE development life cycle means that in order to manually add source code or change the auto-generated source code deliverable files during development, there is the option to regenerate an updated model of application development. This is an important step towards resolving the maintenance issues; however, it cannot be applied in case the generated code originates from multiple models. Additionally, 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 *GrafiXML* [23] or *GuiBuilder* [24], it is practically impossible to recognize the widget elements by parsing manually written source code [25]. Our methodology can be deployed for both general-purpose and mission-specific tools, while still addressing the maintenance issues. In the Figure 11 is depicted the *Papyrus* during construction of a model.

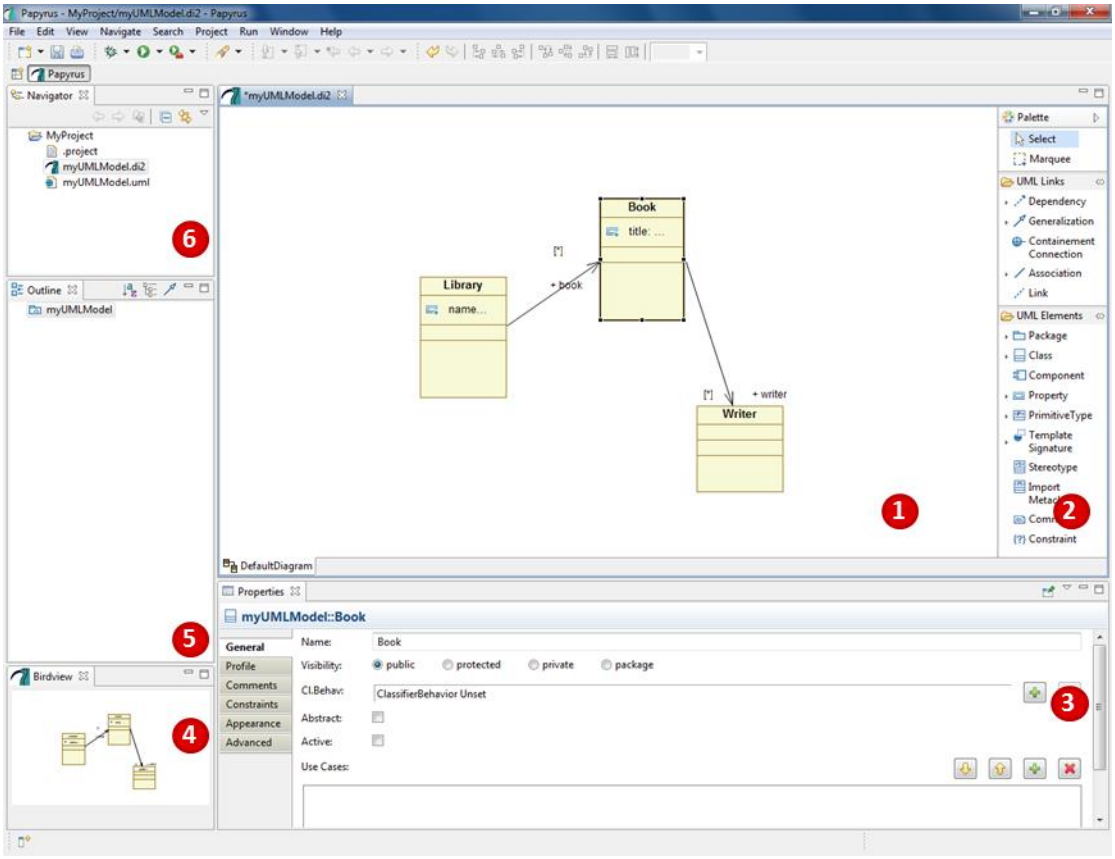


Figure 11. Using the Papyrus; Area 1 is the *"action"* area of model construction; Area 2 is the Palette toolbar of the Model constructs; in Area 3 you can view/edit the data of constructs of models; in Area 4,5 you can see in two different ways the outline of the model; Area 6 is the navigation of project tool.

Modelio

Modelio [26] is an open source modeling environment based on Eclipse. Although Modelio is based on an Eclipse RCP, it is a standalone application, not an Eclipse plug-in. However, Modelio is frequently used in conjunction with Eclipse; both work on the same source code organization. Similarly to papyrus, it offers code generation for a variety of languages and supports the full MDE development cycle thus 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 - as *Papyrus does*. In the Figure 12 is depicted the *Modelio* during construction of a model.

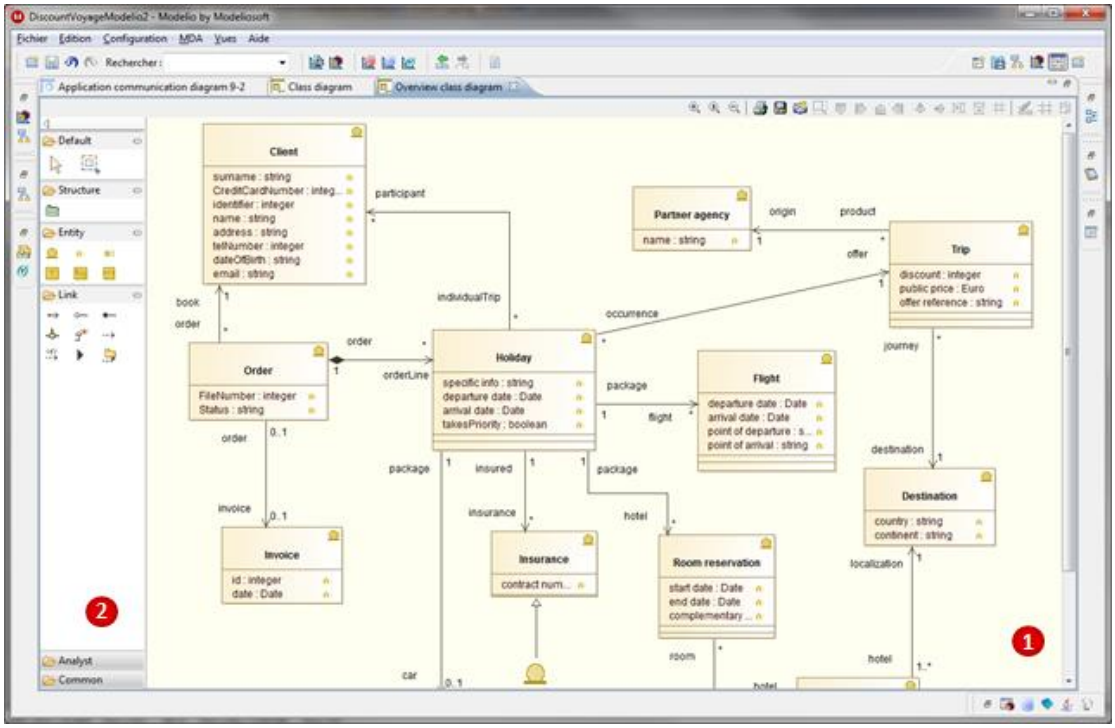


Figure 12. Using the *Modelio* tool; Area 1 is the *"action"* area of model construction; Area 2 is the toolbar of the Model constructs.

Altova UModel

Altova UModel [27] is a commercial UML modeling software tool from Altova. UModel can be integrated with Eclipse and Visual Studio as a plug-in. UModel supports UML 2 diagram types and adds a unique diagram for modeling XML Schemas in UML. *UModel* also supports *SysML* [28] for embedded system developers, and business process modeling (BPMN notation) [29] for enterprise analysts. UModel includes code engineering functionality including code generation in Java, C#, and Visual Basic programming language. *UModel* supports model interchange with other UML tools through the XMI standard, integrating with revision control systems. It also supports reverse engineering of existing applications, and round-trip engineering. In other words, it supports the full MDE development life cycle allowing both model-to-source and source-to-model transformations as the Papyrus and Modelio support. In the Figure 13 is depicted the *Altova UModel* during construction of a model.

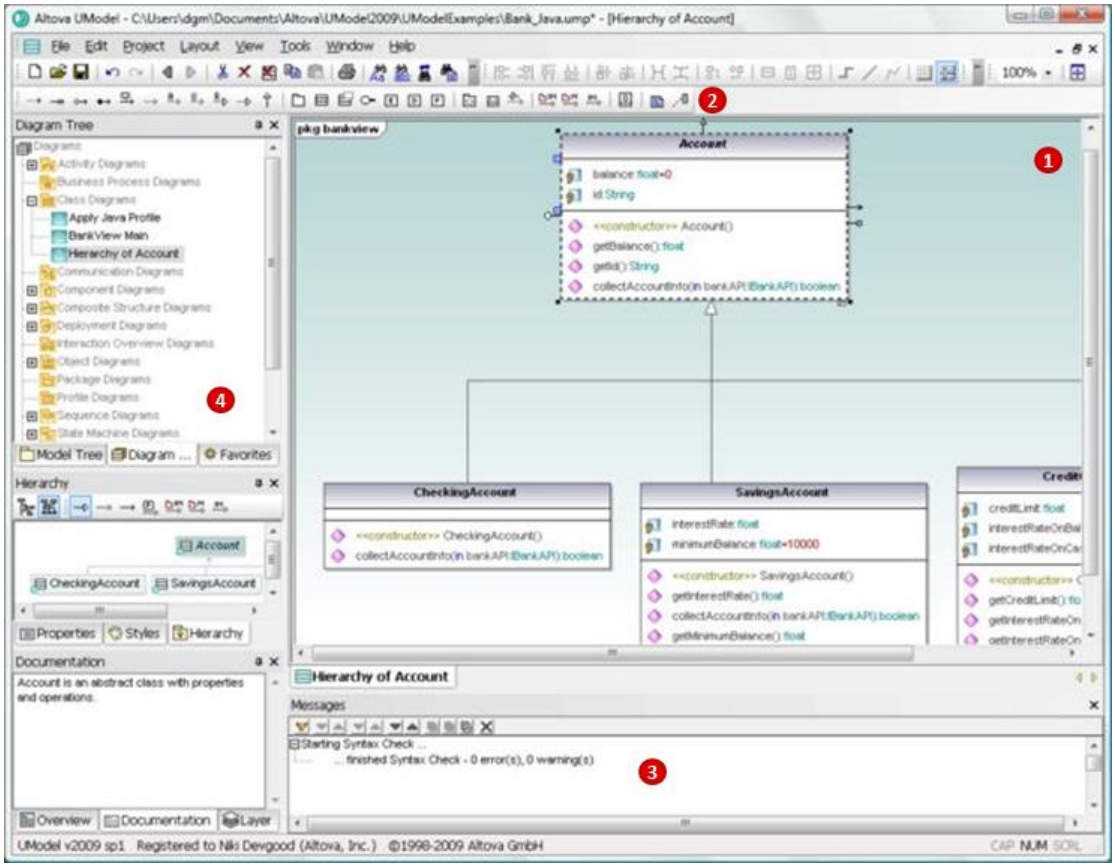


Figure 13. Using the *Altova UModel*; Area 1 is the *"action"* area of model construction; Area 2 is the toolbar of the Model constructs; in Area 3 you can view/edit the data of constructs of models; in Area 4 you can see the Diagram model tree view.

Enterprise Architect

Enterprise Architect [30] is a visual modeling and design tool based on OMG UML from Sparx System. Enterprise Architect supports the design and construction of software systems. It also supports modeling business processes and modeling industry based domains. Enterprise Architect supports code generation in numerous languages like Action Script, C, C#, C++, Java etc. Similar to the aforementioned three tools, it supports the full MDE development life-cycle allowing both model-to-source and source-to-model transformations. This tool, as the previous ones, parses 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. Despite being an indispensable commercial MDE tool employed by several software companies, it still fails to solve the maintenance issue of the specific

mission MDE tool (e.g. GUI builder). In the Figure 14 is depicted the *Altova UModel* during construction of a model.

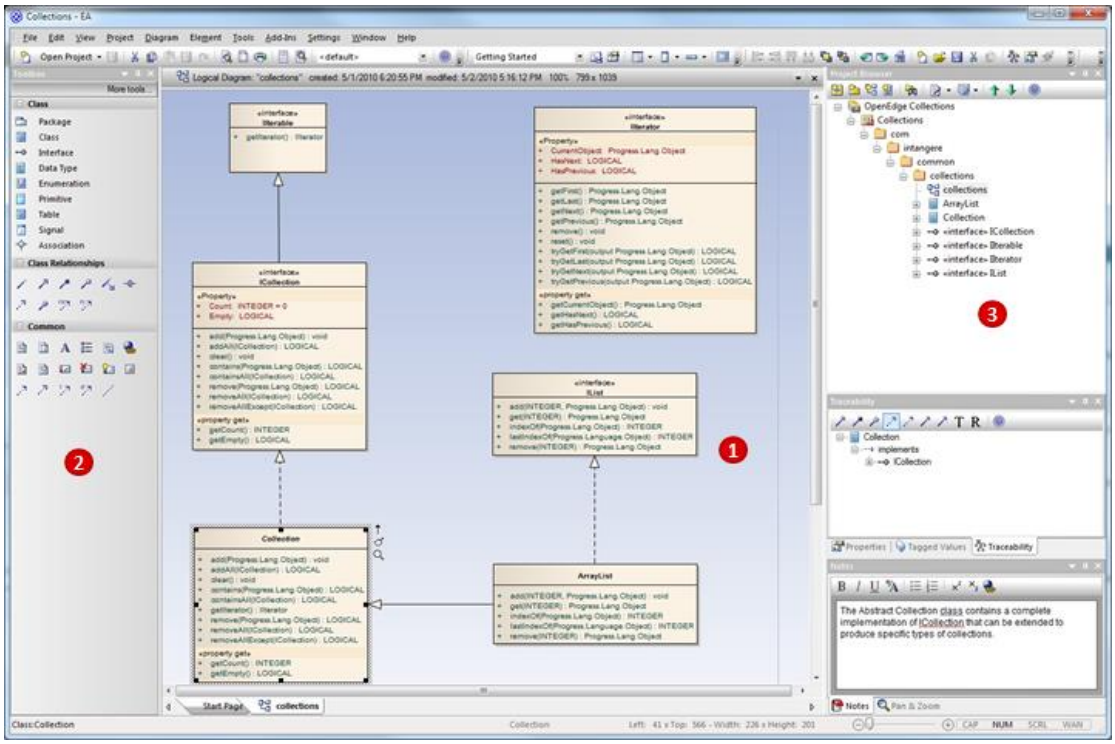


Figure 14. Using the *Enterprise Architect*; Area 1 is the *"action"* area of model construction; Area 2 is the toolbar of the Model constructs; in Area 3 is the project navigation of the tool.

Apollo

Apollo [31] is a robust and flexible modeling extension to Eclipse created by Gentleware AG. Apollo is the first UML extension for Eclipse based on *GMF* [32], *EMF* and *UML 2*, and seamlessly integrates into the IDE. It is available as an RCP stand-alone tool or as an Eclipse plug-in. It is a model-driven tool offering code generation only for Java. Apollo gives developers and programmers the ability to dynamically create and edit models alongside code. That is to say, both model-to-source and source-to-model transformations are allowed which denotes the support of the full MDE development life cycle. In the Figure 15 is depicted the *Apollo* tool during construction of a model.

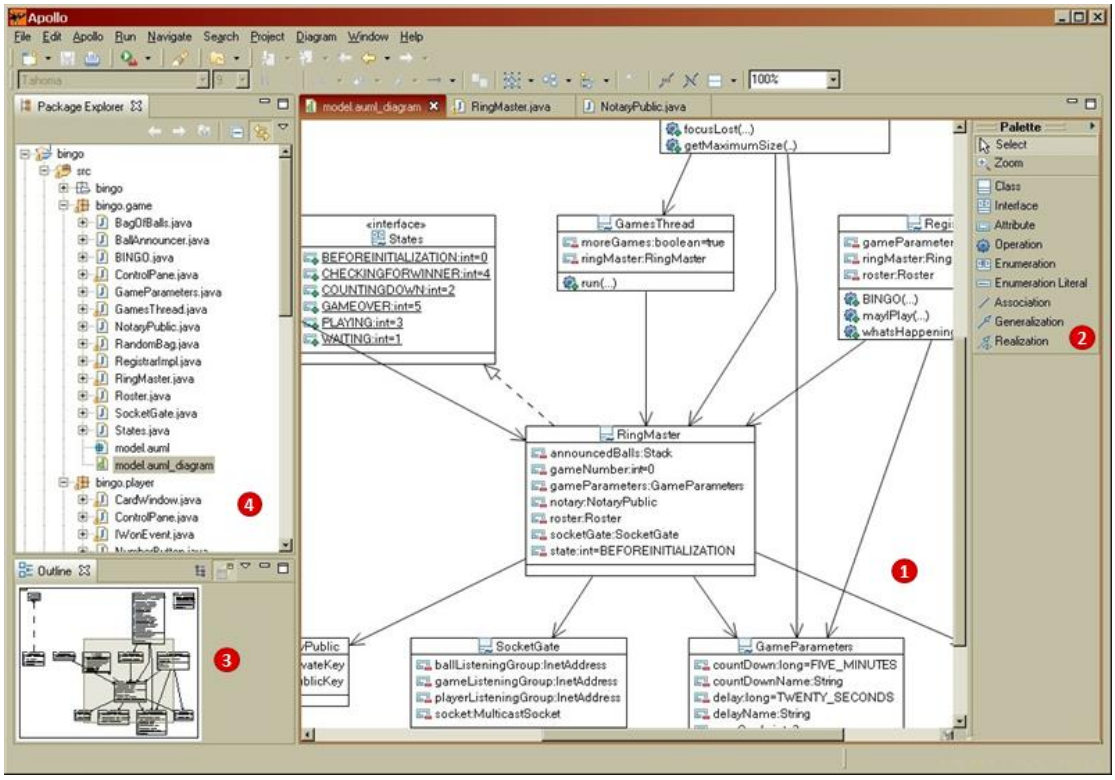


Figure 15. Using the *Apollo* tool; Area 1 is the *"action"* area of model construction; Area 2 is the Palette toolbar of the Model constructs; in Area 3 you can see the outline of Model's diagram; in Area 4 is the project explorer of the *Apollo* tool.

2.2. Specific Mission MDE tools

On the other hand, apart from the general purpose MDE tools and the class hierarchy models of UML, there are MDE tools and description modeling languages which describe a specific purpose of the system under study. As the main category of specific mission MDE tools, we could mention the User Interface Builders. Some of the MDE tools of User-Interfaces are briefly described below. None of them cares for the maintenance issue we address in this thesis.

wxFormBuilder

wxFormBuilder [33] is a popular publicly available interface builder for the *wx* widgets cross platform library [34]. This tool offers a typical rapid-application development cycle with interactive user-interface construction, and outputs

interface descriptions into its custom language-neutral format called XRC [47] (XML Interface Resources). The *wxFormBuilder* also supports code generation of UI for languages C++, Python and PHP. In the Figure 16 is depicted the *wxFormBuilder* during construction of a model.

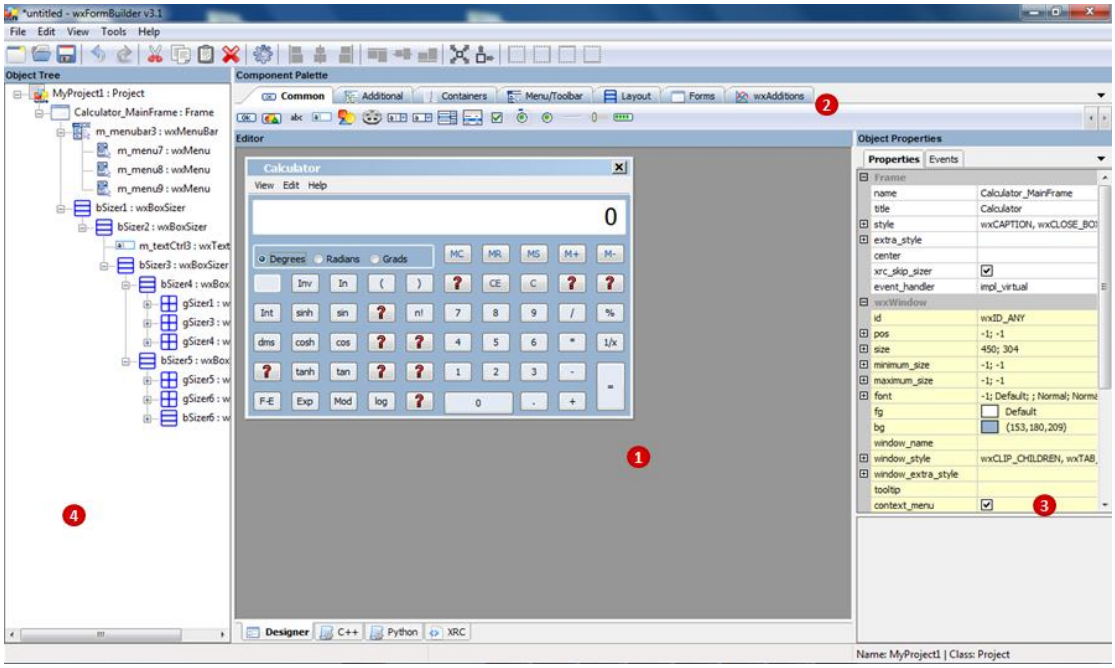


Figure 16. Using the wxFormBuilder; Area 1 is the *editor* of the UI model construction; Area 2 is the widgets toolbar of the UI Model constructs; Area 3 is the view/edit the widgets' properties-events; in Area 4 is the tree view of the constructed UI Model

GrafiXML

GrafiXML is a graphical tool to draw user interfaces. These interfaces could be saved in several formats, like Java or XHTML, but the principal way is to save them in UsiXML [35], an XML user interface description. Then, the final user interface is produced by Rendering or Generative programming. In the Figure 17 is depicted the *GrafiXML* during construction of a model.

.

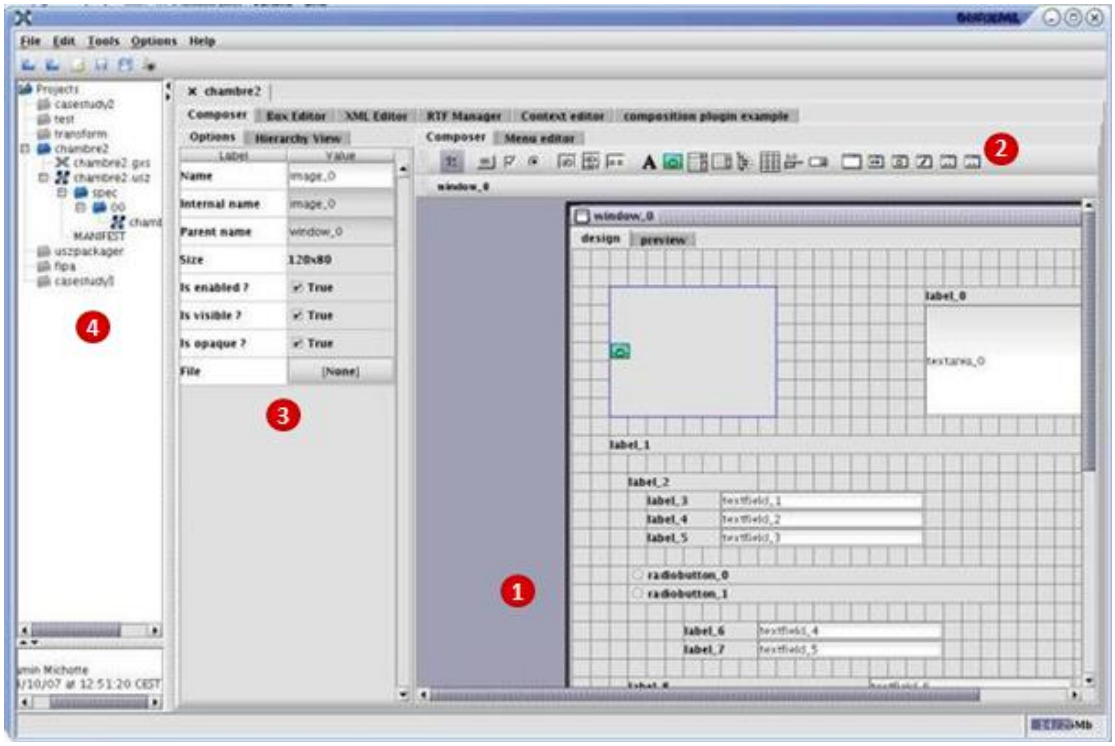


Figure 17. Using the *GrafiXML*; Area 1 is the *editor* of the UI model construction; Area 2 is the toolbar of the UI Model constructs; Area 3 is the view/edit the properties-events; in Area 4 is the project explorer of the GrafiXML tool.

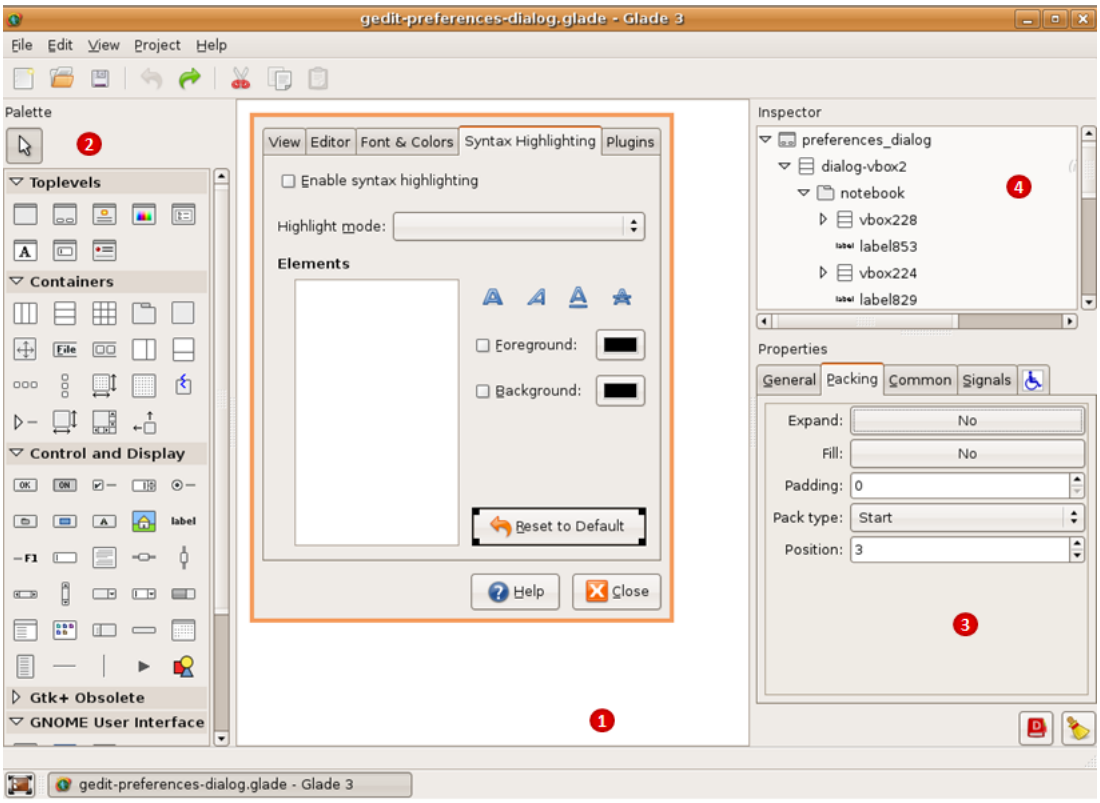


Figure 18. Using the *Glade*; Area 1 is the *editor* of the UI model construction; Area 2 is the widgets toolbar of the UI Model constructs; Area 3 is the view/edit the widgets' properties; in Area 4 is the tree view of the constructed UI Model.

Glade

Glade [36] is a graphical user interface builder for GTK+ toolkit and the GNOME desktop environment. Glade saves the user interfaces designed as XML. Then, using the *GtkBuilder* [37], Glade XML files can be used in numerous programming languages including C, C++, C#, Java, Python, Perl and others. In the Figure 18 is depicted the *Glade* during construction of a model.

wxGlade

wxGlade [38] is a graphical user interface designer written in Python using the *wxPython* [39]. It supports code generation of UI for languages C++, Python, Lisp and Perl. Additionally, wxGlade could generate the User-Interface in the form of XRC (wxWidgets' XML resources). While it is not related to *Glade*, they are similar in idea and in their interface. In the Figure 19 is depicted the *wxGlade* during construction of a model.

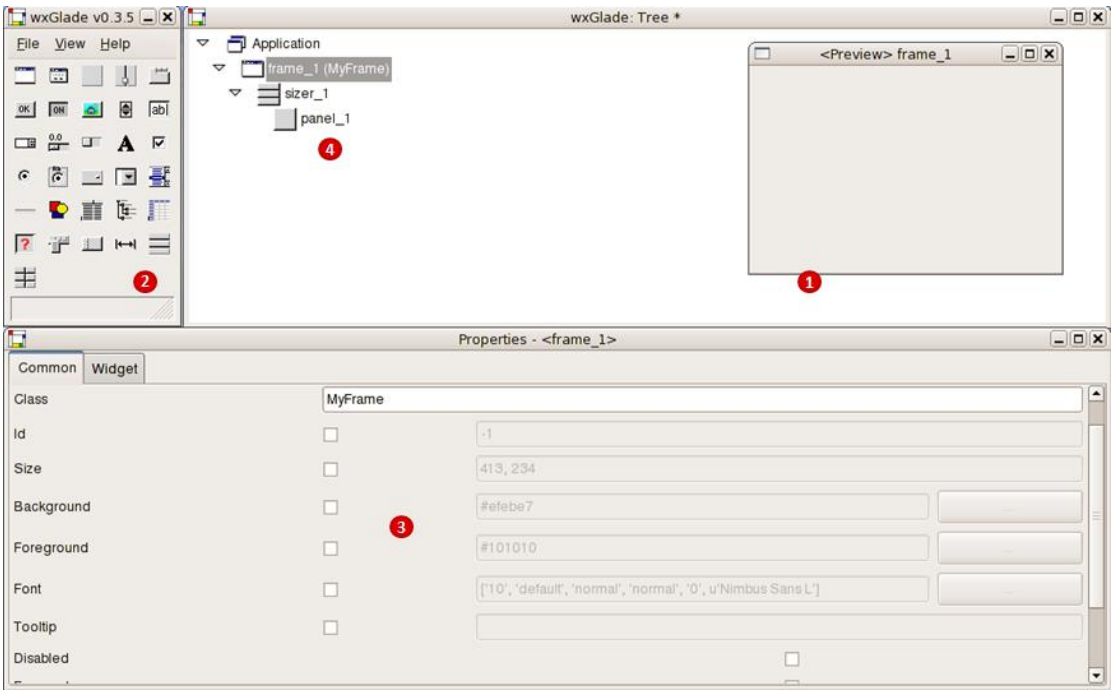


Figure 19. Using the wxGlade; Area 1 is the *editor* of the UI model construction; Area 2 is the widgets toolbar of the UI Model constructs; Area 3 is the view/edit the widgets' properties; in Area 4 is the tree view of the constructed UI Model.

wxDesigner

wxDesigner [40] is a dialog editor and RAD tool for the wxWidgets C++ library. It supports code generation of UIs for languages C++, C#, Python and Perl. wxDesigner could also produce XRC model. In the Figure 20 is depicted the *wxDesigner* during construction of a model.

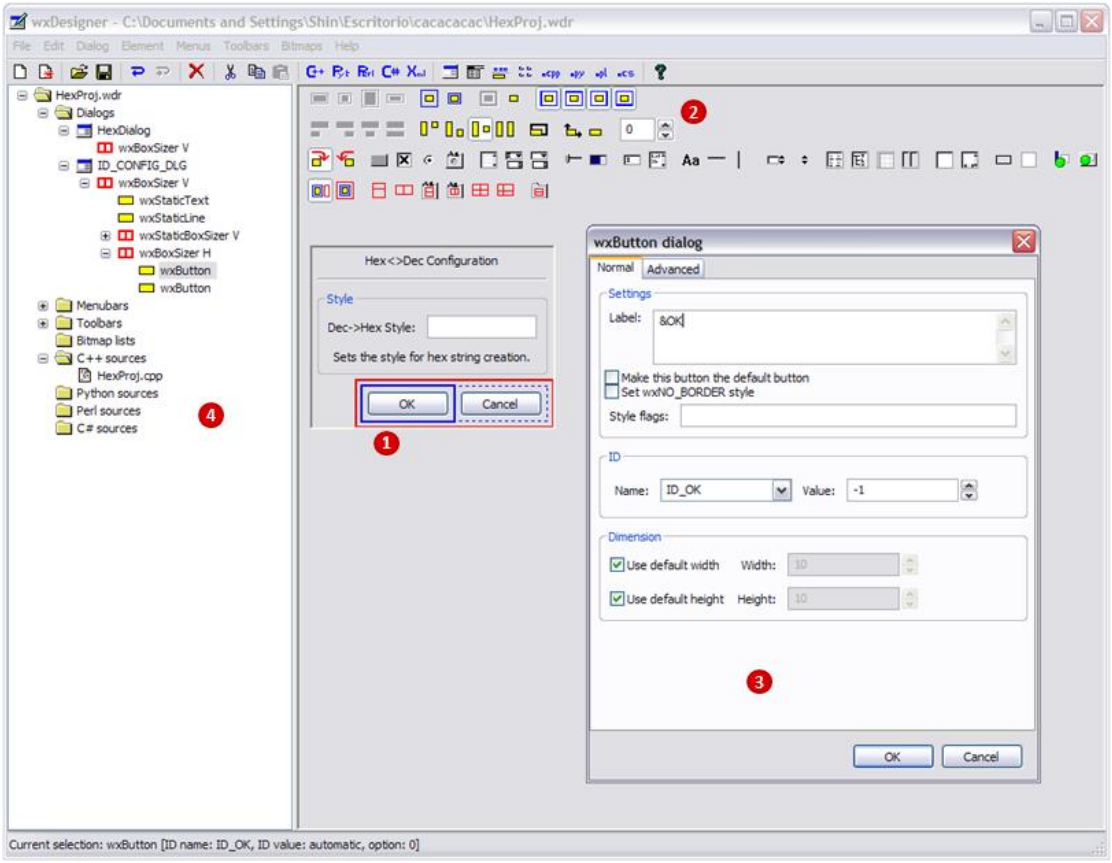


Figure 20. Using the wxDesigner; Area 1 is the *editor* of the UI model construction; Area 2 is the widgets toolbar of the UI Model constructs; Area 3 is a view/edit dialog of widgets (opening when double click in the widget in the editor); in Area 4 is the tree view of the constructed UI Model.

Blend

Blend [41] is a User Interface design tool developed by Microsoft for creating applications' graphical interfaces for desktop and web. It is an interactive, WYSIWYG front-end for designing XAML [42] -based interfaces. In the Figure 21 is depicted the *Blend* during construction of a model.

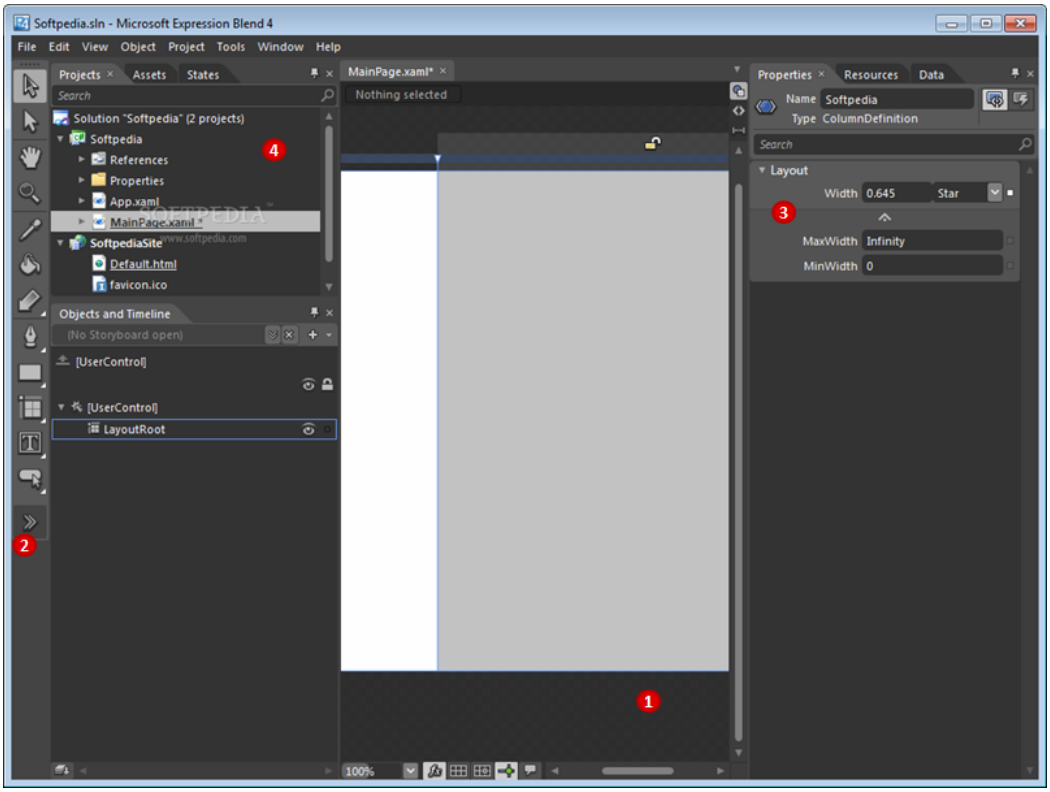


Figure 21. Using the *Blend*; Area 1 is the *editor* of the UI model construction; Area 2 is the toolbar of the UI Model constructs; Area 3 is a view/edit widgets properties; in Area 4 is the project explorer of the *Blend* tool.

Additionally, there are MDE approaches for networks. For example, the *Analysing Wireless Sensor Networks* [43] in which there is the *WSN Modeling Languages* and then code generation using *Acceleo* which we described above and which does not solve the maintenance issues. There is also the MDE approach that provides resources to non-specialists in parallel programming to implement their applications [44]. In particular, it provides code generation from UML/MARTE to openCL. In this case, a description language is used too and uses the Acceleo for the code generation similar to the approach of the Network described previously.

Chapter 3

Improved Process

In this chapter we are going to describe the improved process of Model-driven engineering using metaprogramming. The proposed methodology, illustrated in Figure 22, consists of 3 main steps.

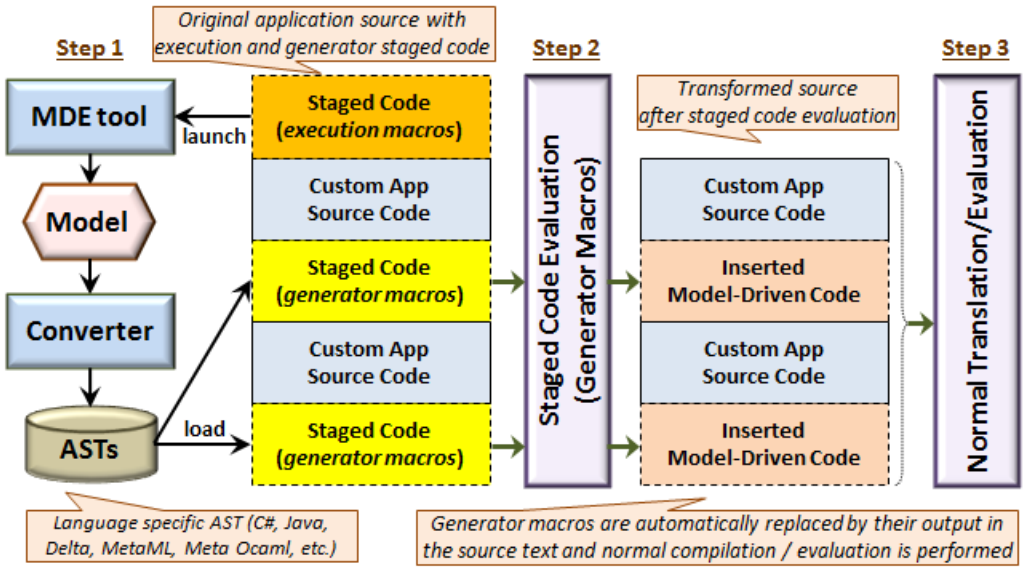


Figure 22. Encapsulating the model-driven process directly in the application source through staged metaprograms. *Step 1*: Staged code execution macros invoke the MDE tool that creates the model and converts its corresponding code as ASTs. *Step 2*: Staged code generator macros take the ASTs as input and insert the model-driven code into the source along with custom application code. *Step 3*: The transformed source is normally translated or evaluated to produce the final binary of the entire application.

Firstly, the staged code contains execution macros responsible to externally launch the MDE tool (Step 1). Then, we deploy a converter to turn the model entities into source code fragments stored in AST form. Afterwards, we manipulate the ASTs in order for them to be ready for deployment. The generated ASTs are then loaded by the staged code generator macros and insert the model-related code into the source along with custom application code (Step 2). Finally, the transformed source resulting from the staged code evaluation is normally translated or evaluated to produce the binary image of the entire application (Step 3).

In the following subsections, we continue with the analytic description of the steps of the proposed MDE process.

3.1 Tool Chain

In general, the first step of the Model-driven Engineering development is to create one or more models. Then, it continues with model-to-model transformations, simultaneously decreasing the abstractions of the models and approximating the real system. Afterwards, the model to code transformation is applied and the developer completes the system that needs to be finalized with manually written source code. In the development's life cycle, it is very common to decide changes for one or more models of the system under study. In this case the whole process described above needs to be repeated. All this development life cycle demands the use of MDE tools, in order to handle the models conveniently and effectively. In our work, we focus on MDE tools generating source code, either entirely or partially. So, in this section we will go on to describe the invocation of MDE tools and the deployment in our approach.

3.1.1 Invocation

The Model Driven process begins with the invocation of MDE tools in order to construct a model which describes the application. The invocation of this MDE tool can be done by employing two different ways:

The first way is the invocation of an external MDE tool to construct the model, produce the correspondent auto-generated source code. Then the IDE is opened in order to handle the auto-generated source code and also develop the manually-written code. As discussed previously, it is common to edit the model and regenerate the source code numerous times in the development life cycle. This shows us that it is not really effective to use an external tool in combination with the IDE during development.

The second way provided, is the invocation of MDE tools included as plugins in the IDE. In the one hand, this solves the inconvenience of the use of an external MDE tool but on the other hand reduces the choices of MDE tools used in development. We now proceed to describe an alternative path in the invocation-use of MDE tools, focusing on bringing the MDE deployment as close as possible to the actual application development.

Invoking MDE tools through metaprograms

The use of generative MDE tools typically involves first launching the tool, secondly loading or creating a model, then performing any necessary modifications on it and finally generating its corresponding code that will be used as a basis for the entire application development. The target of this entire process is always to obtain the generated code: the MDE tool is typically not launched again unless the model needs to be updated, while any model updates result in code regeneration. In the latter case, the final application code also needs to be rebuilt to reflect the latest model changes.

Since eventually the desired effect is to link the latest model code with custom application code, it is possible to invert the MDE tool deployment as follows; whenever the application is to be (re)compiled, if any changes need to be performed on the model, we launch the tool, perform the necessary updates, regenerate the code and finally compile it along with the remaining application code. This observation has led us to the idea of utilizing staged metaprogramming as a method for orchestrating the MDE deployment directly through the original program source.

The staged code contains execution macros responsible for launching the MDE tool. Once the tool is launched, the developer may normally interact with the model, updating it as needed. Then, the process continues with the model-to-text transformation based on the updated model. Afterwards, the compilation of the system continues normally.

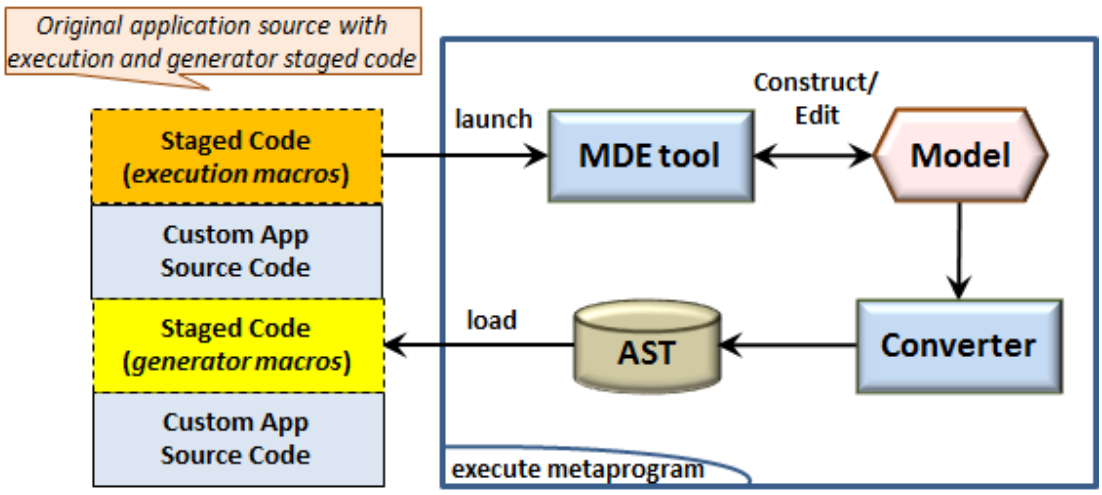


Figure 23. Invocation of MDE tools in the beginning of build process through staged code.

As illustrated in Figure 23, the development life cycle begins with the development of the correspondent staged code with execution macros in order to invoke the MDE tool during the compilation process. Then, in the first compilation of the main program the MDE tool(s) are invoked in order to construct the model(s) of the application under study. Afterwards, in case any changes need to be performed on the model(s), we launch the MDE tool(s) during the compilation of the application.

The advantage of this process is that it provides the ability to invoke any MDE tool externally just with the correspondent execution macros of the tool during compilation of the system. In other words, there is freedom to conveniently use during development time whichever MDE tool we need for the development of a system and not be based on specific MDE tools that may be included as plugins in IDE. On the other hand we have to note that to invoke an external MDE tool developers must be knowledgeable of the relevant system-call command of the staged code which can run the correspondent MDE tool which will simultaneously launch the chosen model. Certain tools lack this type of system-call commands. There are for example tools that only provide available commands to load their project file and not the model (e.g. *wxFormBuilder* which is used for case studies).

Using staged metaprogramming for the invocation of an external MDE tool, gave us the additional idea of an alternative way to update models without the need for external MDE tools. This approach focuses on implementing the model editor as an inherent part of the metaprogram i.e. without launching any external applications. This way we bring the MDE deployment to the actual application closer than previous approaches we discussed above. Of course, such a custom editor need not be implemented from scratch but may reuse any model editing library implemented in the same language. Using this approach we may take advantage of executing in the same address space and also store the generated data in a metaprogram variable that can be later used directly in the generator macros, thus removing the need for reloading the data from storage. Additionally we have to note that, the implemented model editors can be used from the beginning of the model-driven process in order to construct the model(s) of application. They only need to run them separately us a program inside from IDE.

3.1.2 Deployment

After using the generative MDE tools in order to construct the model, the next step is the auto-generation of the correspondent source code. This auto-generated

deliverable has to be adapted in order to complete the development of application. In case developers create one model for development, they have to use the auto-generated deliverable created by model and combine it with their hand written source code in order to complete the application's source code.

In general, during the development process it is common to use more than one model-driven (MDE) tool to construct a single application. Each MDE tool is used to construct one or more models (Step 1 in Figure 24). Source code fragments are produced for each of these models by correspondent code generators. Afterwards, developers have to combine these source codes in order to complete the development process (Step 2, 3 in Figure 24).

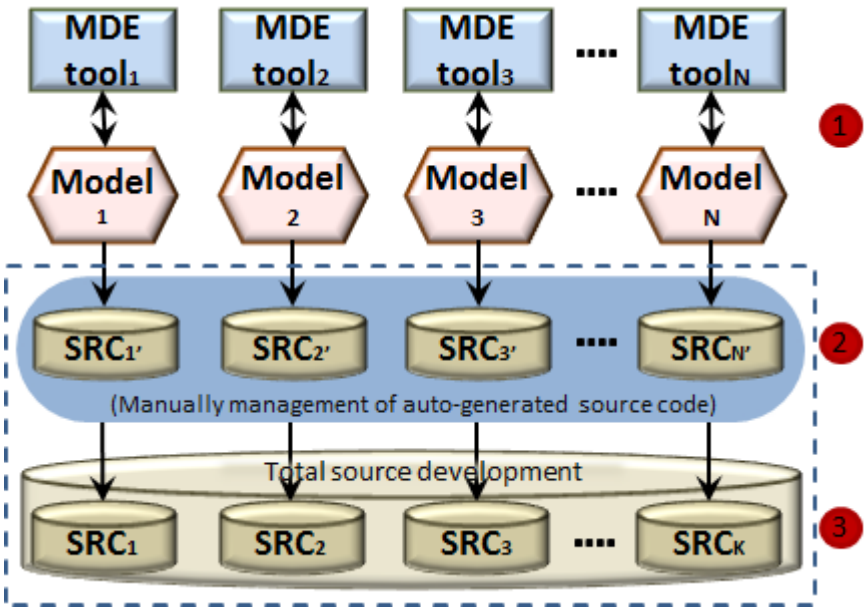


Figure 24. Deployment of MDE tools in the development process

During the development process, there are a lot of times when there is a need to edit models (Step 1 in Figure 24). These models can be edited in order to complete the development or to change something in the developing system repeatedly until the end of its development. Each updated model demands the regeneration of its correspondent source code. The maintenance issue is now a problem not only between the developer's code and the auto-generated code, but also between the auto-generated source codes. In other words, developers have to repeat Step 2 and

Step 3 that is depicted in Figure 24, for each updated model of the developing system and all auto-generated codes that have dependencies from it in auto-generated sources. Instead of this approach, we propose the use of metaprogramming as we mentioned previously in the introduction of this chapter. In other words, we develop programs (staged code) which manage all the auto-generated deliverables. Firstly, programs load the model and produce the auto-generated deliverable with appropriate converters. Then, programs edit/extend the deliverables and finally inline whole or parts of the deliverables between the source code of the custom application.

3.2 Producing ASTs

In general the Model Driven Engineering tools get a model as input, or construct a model and deliver other models or source code. In other words model to model and model to text transformations are applied. The last step of this process would be a model to text transformation. Before the model to text transformation happens for the last time we need to update the models and repeat the transformations while any manually written source code has been added in the auto-generated source code. So, 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 staging into the pipeline.

In particular, with staged model driven generation the MDE 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 the code-to model reconstruction path is unnecessary.

3.3 Transforming ASTs

Using MDE generative tools which produce source code from model(s), developers have to complete and transform the auto-generated code in order to finalize the system under construction. In our case, we produce AST instead of source code as mentioned in the previous section. This means we have to handle the ASTs in order to transform their contents which are a tree representation of the abstract syntactic structure of source code written in a programming language. The generator macros may contain any application-specific composition or editing logic. Practically, this means that it is possible to perform any code transformation on a source fragment before inserting it in the final source. There are two different places these transformations could be deployed. The first place is in one or more separate programs. This way is described in section 3.3.1. The second place is in the source code of the development application with embedded staged metacode. This way is described in section 3.3.2.

3.3.1 Batches - Separate Metaprograms

In our approach, it is proposed that the MDE generative tools produce ASTs - instead of source code - which are saved in the form of binary files. In the development's life cycle, the auto-generated deliverable(s) needs a lot of changes and additions of source code in order to complete a software system. In our case, we can separate the changes or additions by specific programs (e.g. write a program in a script to add the event connections to a User-Interface AST). In other words, after

the model to AST transformation, AST to AST transformations are deployed. Separate programs are run which load the AST from a binary file, deploy the transformations and save it in a binary file as depicted in Figure 25.

Following this approach, we gain the flexibility and reusability of the transformations of AST, and additionally the maintenance problem does not reappear. In case developers decide to change or add to a metaprogram then they do not need to run the whole sequence of metaprograms from the beginning. They only need to run the altered metaprogram and those following it. In other words, if we had changed the Metaprogram2 as it is depicted on Figure 25 we would not have to run the Metaprogram1.

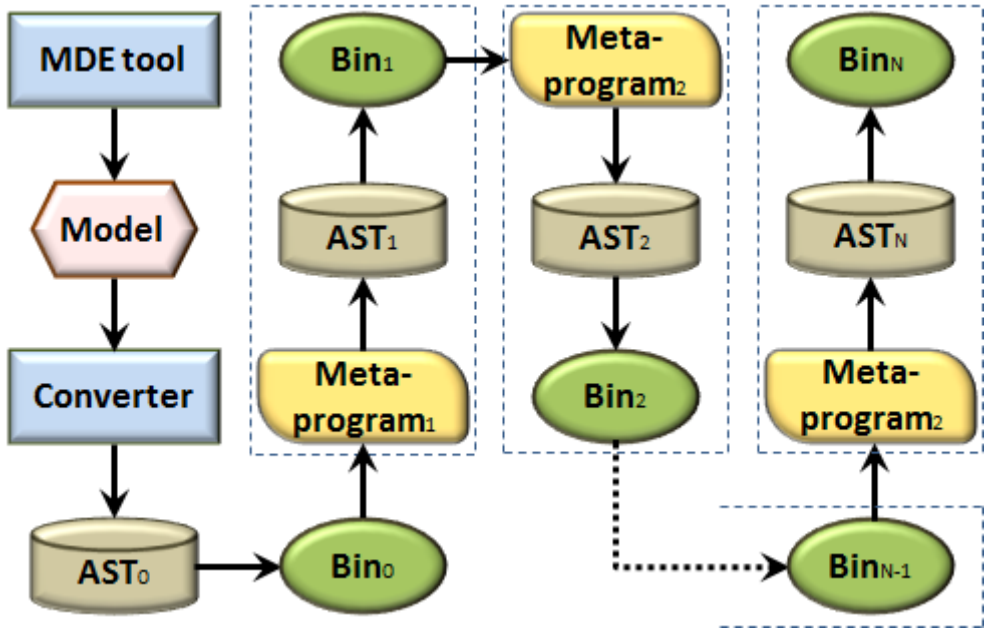


Figure 25. Running meta-programs which load the binary files of AST; transform and save it back to the disk.

Using this approach, it is easier to reuse the metaprogram AST transformations for different ASTs and to debug these programs' correctness. These AST to AST transformations are in the one hand language dependent and on the other hand it approximates the source code of the system without necessarily being part of the application development. In other words, we could deploy separately the process of

editing the auto-generated deliverable of generative MDE tools from the development process.

3.3.2 Stages - Embedded Metaprograms

Additionally, one other place to transform the auto-generated AST is in the source code of the system under construction with embedded metaprograms. In these metaprograms has again to load the ASTs and then deploy the transformations. When the compilation of the system under construction starts the metaprograms first build and then runs during compile time, as result, loads the ASTs and executes their transformations. In case of run-time metaprogramming, firstly is built total source code and meta-code. Then, during run-time first execute all the meta-code and runs the system after that.

Choosing this approach in the one hand, all the transformations will be done in the build process of application development and there is no need to save binary files in the disk as all the AST's transformations save in the memory of the program at compile-time; on the other hand there is no segmentation of the AST transformations to check their correctness and there is no reusability of the meta-code in case you want to reuse the meta-code for other ASTs.

3.3.3 Combining Batches and Stages

Finally we have to note that there is no restriction in using both Batches and Staged AST transformations. We could develop separate programs in order to deploy AST to AST transformations and then develop embedded metaprograms in order to deploy other transformations. The latter could possibly be more specific AST transformations for the development of an application. Additionally, separate programs could be added as meta third-party libraries in the application and called in

the embedded metaprograms in order to do the transformations in one entire process.

3.4 Unparsing ASTs

After AST to AST transformations have completed, as next step and last of proposed approach we have to generate the source code from ASTs. In order to do this, we have to add embedded metaprogram which begins by loading the auto-generated AST (this needs in case used only batches for the AST transformations). Then, we have to add embedded metaprograms in order to place the auto-generated deliverable in the manually written source code. This is based on the operators of metaprogramming which are offered by the used language. Embedded metaprograms can be placed everywhere among the source code. Consequently, parts of ASTs could be placed anywhere in the manually written source code.

The compilation result of the staged code incorporates the source code inserted by AST with the manually written source and constitutes the final source code of the application as shown in Figure 22. The final source code is created during the end of the compile time process for the staged source code. The final source code is the combination of the manually written source code and the auto-generated source code, and is read-only. Finally, after the staged evaluation has produced the final source code, the process continues with the normal translation (compile-time staging) or evaluation (runtime-staging). In case of the run-time metaprogramming, the staged code is run first and then the system source code. In this case too, there are no maintenance issues.

This way of generating source code from ASTs changes the model-driven process of generative MDE tools. During the refined model-driven process with an inverted responsibility through staging, programmers deploy generator macros to insert generated code on-demand and in-place without affecting the originally produced ASTs by the MDE tools (see left part of Figure 26). This substitutes the process of

transforming the auto-generated source code files in order to complete the application development (see right side of Figure 26).

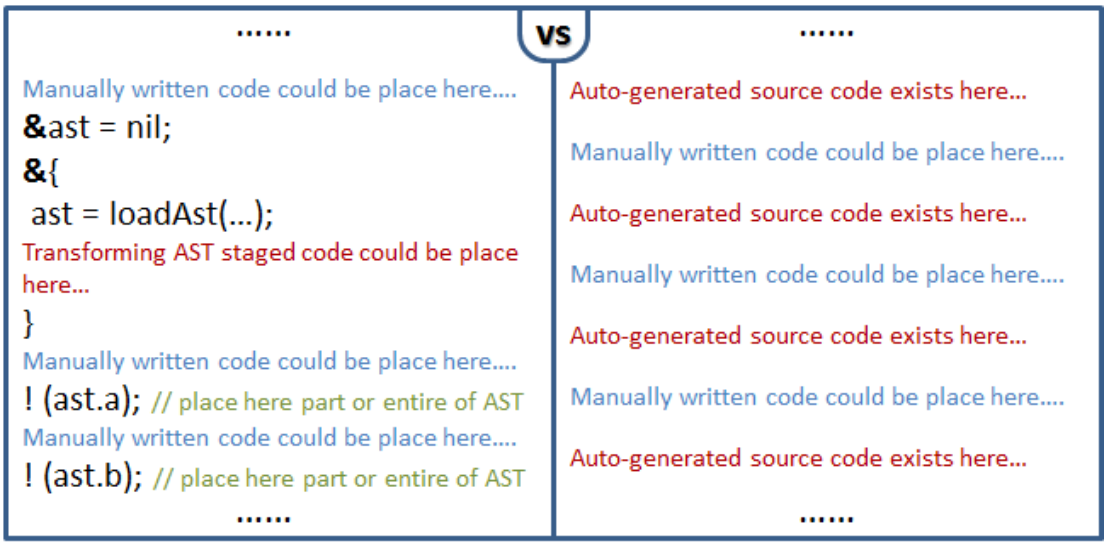


Figure 26. In the left part, we can see a form of our approach source code; in the right part the source code of a classic model-driven process.

This approach may look more difficult than the classic approach but as we discussed in a previous section the only thing we have to do in order to write source code in the form of AST is to add **<<** … **>>.** In the next chapter we deploy this proposed approach and its effectiveness will be clearer.

Chapter 4

Case Studies

To test the proposed MDE approach of our work and assess the expressive power and its engineering validity, we applied several case studies. In this chapter we described them by separating them in four different categories. Each of these categories is a case that could arise in the development of a system. In the first section, we describe cases in which we have to develop an application using a tool to construct a model of the User-Interface. In the second, we outline the case to construct a class hierarchy model for an application. Afterwards, we describe an alternative way to define a model by specifications to auto-generate a User-Interface application in respect of our approach. Finally, in the last section we describe the case of using more than one model to construct a single application.

4.1 User Interface Builder

As we mentioned in section 3.1.2 we can deploy our approach for a separate tool or with a combination of tools. In the next subsection, we describe the deployment of the approach, focusing on User Interfaces. Additionally, to test our approach and assess its expressive power and engineering validity, we have carried out case

studies which are described in section 4.1.2. In particular, we have developed a full-scale scientific calculator application, a paint basic application and finally we developed the Self MDE deployment (part of our MDE approach). We continue with the description of the User-Interface deployment of our approach and then following with the case studies.

4.1.1 Applying our approach for UIs

In the beginning, we deployed our approach focusing in User Interface Builder as it is outlined in Figure 27. We had to use a specific interface builder which delivers a specific User Interface Description Language [46] (UIDL) model. So we have adopted a WYSIWYG tool, the wxFormBuilder [33], a popular publicly available interface builder for the wx widgets cross-platform library. This tool 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). Then, using wxFormBuilder we construct application and get as output the correspondent XRC model. To convert XRC to the Delta language ASTs, we developed an appropriate converter. Then, using the metaprogramming features of the Delta language, we import and manipulate the application ASTs, and also add extra interactive features and behavior to it, besides the ones introduced merely with the wxFormBuilder.

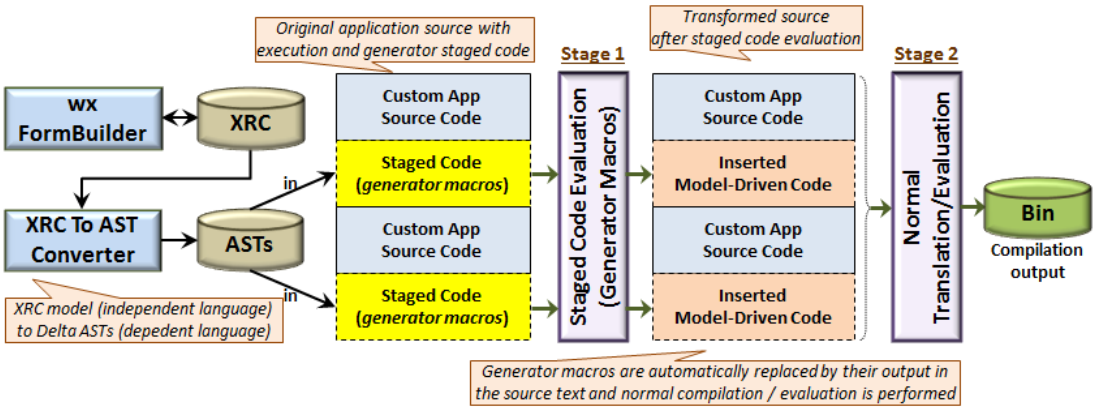


Figure 27. Deployment of approach focusing on User Interface builder.

So, by designing the simple dialog which is depicted in label 3 of Figure 28 and developing the source code which is outlined in label 2 of Figure 28, we have the resulting read-only source code which is depicted in label 3 of Figure 28 when the built process finishes.

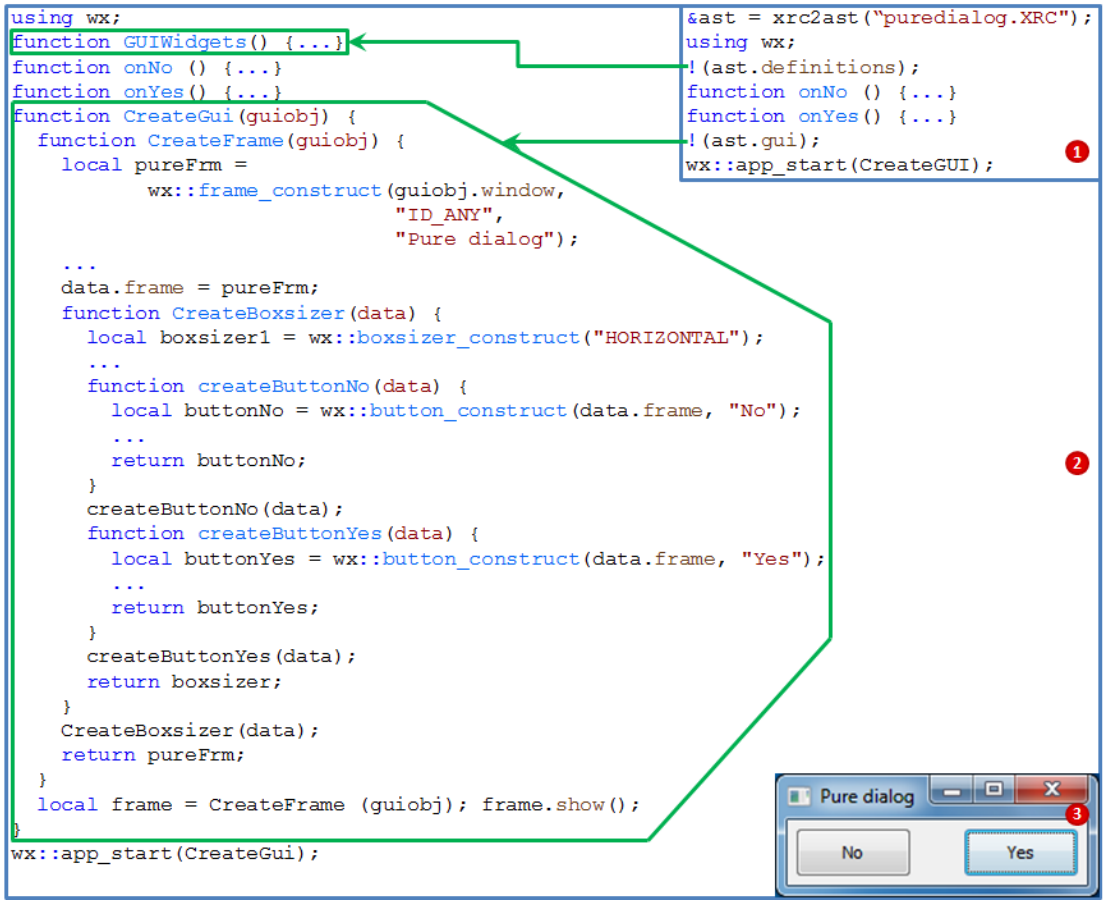
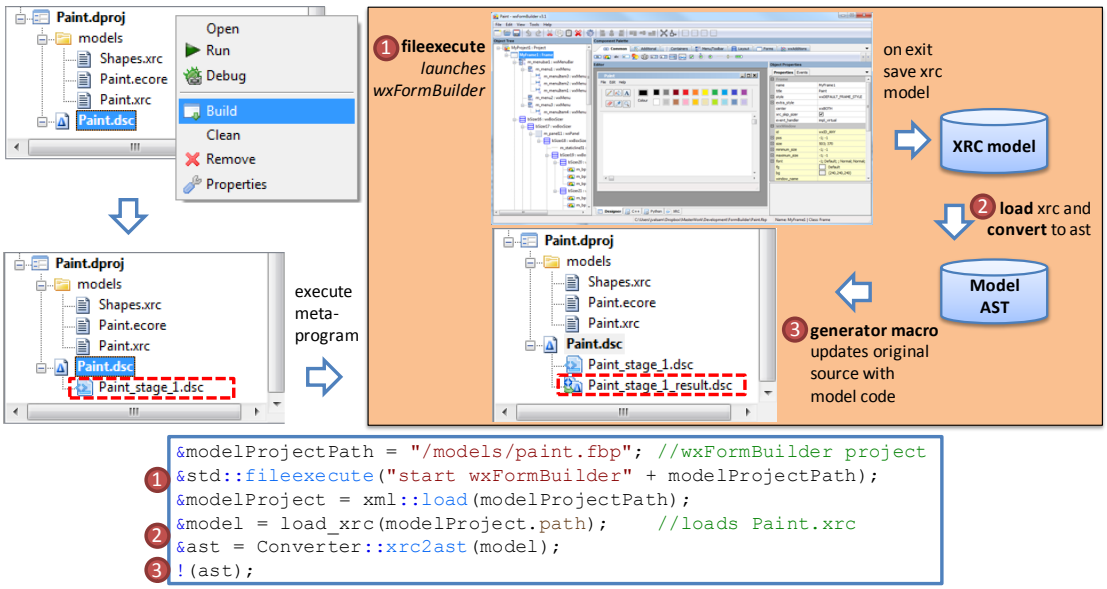


Figure 28. Code generation; in label 1 the application for a pure dialog code with embedded staged code is outlined; in label 2 there is the result of the build process (i.e. the generated code which is read-only); in label 3 the pure dialog is depicted.

Invoking UI Builder through a metaprogram

Additionally, using the metaprogramming features of the Delta language, the wxFormBuilder was launched directly from the meta-code during compilation to allow interactive editing of the user interface. The entire process is illustrated in Figure 29. In particular, during the compilation of the target application, e.g. a Paint application, we assemble and compile the stage metaprogram, i.e. Paint_stage_1.




```
&modelProjectPath = "/models/paint.fbp"; //wxFormBuilder project
&std::fileexecute("start wxFormBuilder" + modelProjectPath);
&modelProject = xml::load(modelProjectPath);
&model = load_xrc(modelProject.path);     //loads Paint.xrc
&ast = Converter::xrc2ast(model);
!(ast);
```

Figure 29. Overview of the compile-time MDE deployment through staged metaprogramming. Actions performed during the metaprogram execution (top right) and their corresponding source code lines (bottom) are shown with matching numbers

Then, during the metaprogram execution, the call to std::fileexecute launches the wxFormBuilder with the specified model as input (step 1). The metaprogram execution will suspend until the call to std::fileexecute returns, something that occurs only after closing the launched application. When the interactive editing is completed, the XRC model is saved, the wxFormBuilder is closed and the metaprogram resumes execution by loading the updated model and converting it to AST (step 2). The latter is then inserted into the program source through a generator directive (step 3) and the transformed main program, i.e. Paint_stage_1_result, is normally compiled to produce the final application.

Manipulating User Interface Code as ASTs

The goal of our case studies in User Interface Builder category is dual: (a) to show that the maintenance is effectively eliminated; and (ii) to demonstrate the huge expressive power of metaprogramming for flexible interface code composition. In this context, as part of the case study, we have identified and deployed a number of operations on ASTs to assist in code composition when implementing user-interface metaprograms. The notion of user-interface code is not limited to user interface construction logic, such as creating widgets and setting their visible and layout properties. It actually concerns the full range of dialogue management requirements, including event management and all types of dynamic interface updates. For instance, composition may well concern scenarios were event management code is injected within a user-interface construction code snippet.

Next we continue by enumerating and briefly discussing the manipulation operators. A few automations for easier user interface code composition were provided on insertion, such as renaming of local variables in case of conflicts at the new context, and automatic relinking of widgets to the container produced by the most previous code fragment.

Clone

Concerns cases where a copy of the source code for a user interface component is required. Typically, alone this operation is rarely needed, thus it is anticipated to be followed by radical changes of the user-interface code with operations such as merge, insert and modify.

Cut

Addresses the need to extrapolate the code snippet of an entire user-interface component, and is expected to be followed by appropriate merge or insert operations.

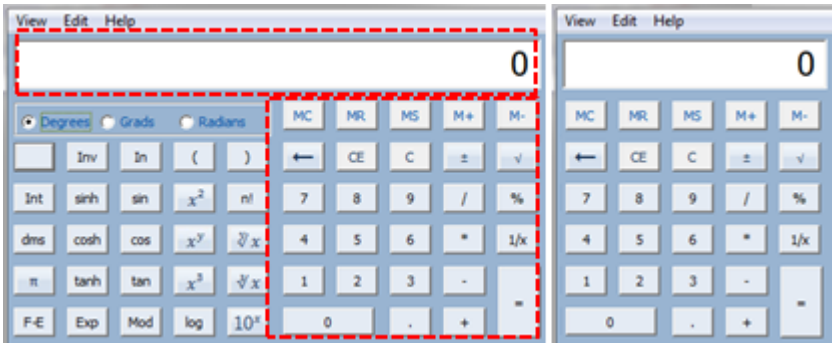


Figure 30. Cut UI parts of Calculator in order to transform scientific calculator in a simple calculator

Crop

It is required when the source code creating some outer parts (i.e. containers) of user-interface components is not needed. In our case we deployed the operator to drop the containing frame window that is by default inserted by the wx Form Builder on all projects.

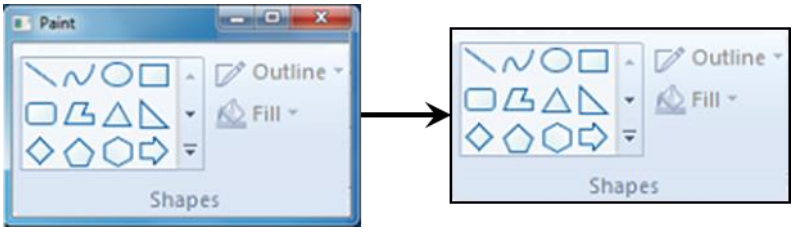


Figure 31. Crop the auto-generated frame from Shapes' toolbar User-Interface.

Create

It reflects the necessity to introduce extra custom user-interface source code in the form of AST, to be actually combined with the parts produced by the MDE tool. In our case we created functions which name begin with the prefix *"createast_"* and continues with the correspondent name of widget as name of create function. Each function constructs the correspondent widget's AST. So, developers have the ability to use these functions instead of *wx-widgets* when they construct user-interfaces with the proposed approach. Each function gets the analogous inputs which usually the widget constructor includes and maybe AST internal body (e.g. panel, sizers etc.)

to insert the AST of children in their body. Finally, returns the appropriate AST. Additionally, widgets' AST can be produced without use of these functions. Using the *quazi-quotes* (<<>>) the widget's AST can be produced. A typical example of *Create* is in the Figure 33. As it is depicted, firstly it is created Shapes' UI toolbar in the appropriate AST Code and then Shapes' UI toolbar is inserted in the Paint's application UI.

Merge

It is a combined composition action on ASTs and is introduced to enable mixing of independent interface code snippets under a common parent. Usually, such components are either authored independently in the modeling process, or they may constitute the outcome of earlier cut operations. A typical example of *Merge* is depicted in the below figure.

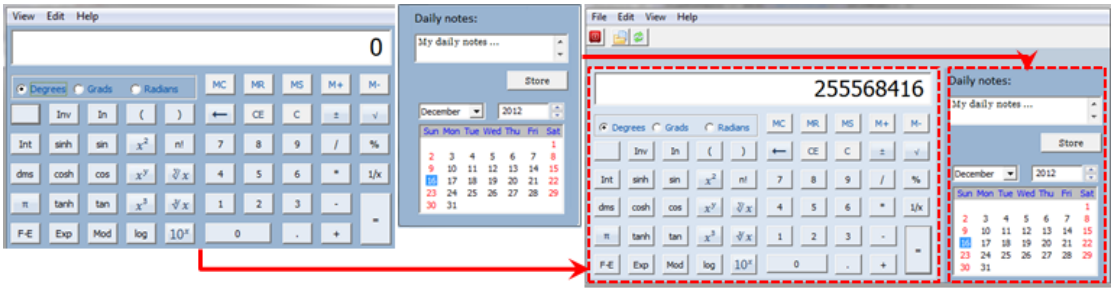


Figure 32. Merge two independent UI code; Calculator and Calendar UIs in one UI application.

Insert

It allows (re)linking of an existing user-interface code fragment inside another one. Practically, this action is the dynamic form of all manual editing actions that user interface programmers would have to apply in order to insert custom code inside the generated code. It is anticipated as the most frequent editing operation on ASTs. A typical example of *Insert* is in the below figure. As it is depicted, in the paint application is added a toolbar with Shapes.

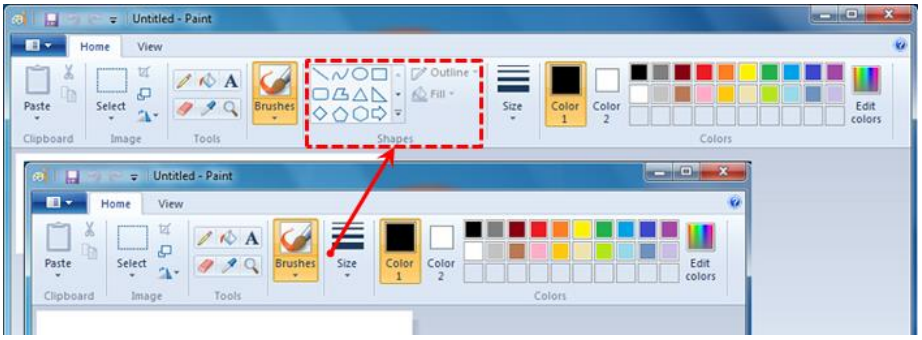


Figure 33. Create & Insert Shapes' User-Interface toolbar in the Paint's application User-interface.

Modify

It reflects the need to algorithmically apply localized changes on the AST, such as: renaming variables and functions, changing argument ordering, changing invocation styles, etc. Although expected to introduce small scale changes, it can be very useful to keep the generated code synced with newer versions of widget libraries when the MDE tool is not yet up-to-date.

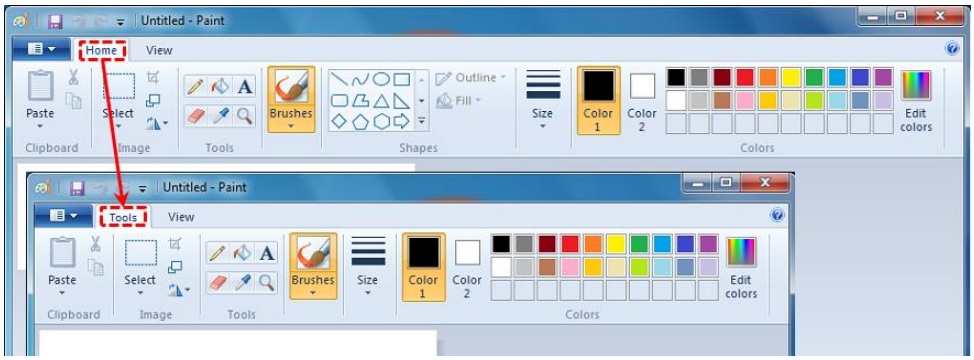


Figure 34. Editing the tab's text "Home" of the Paint's application User-Interface.

4.1.2 Developing User Interfaces

Scientific Calculator

Using the deployment for User Interfaces of our approach (4.1.1), we have carried out several Case Studies. In this Section we describe the case study of a full scale scientific calculator.

In the beginning, we constructed an XRC model of a calculator application using the wxFormBuilder. The latter was actually practiced in alternative ways, such as with single authoring project or alternatively with multiple independent projects. This way we could also assert the compositional flexibility of our proposed approach in combining independently authored interfaces under a single coherent interactive system. To convert XRC to the Delta language ASTs we used the appropriate converter we developed. Then, using the metaprogramming features of the Delta language, we imported and manipulated the calculator ASTs, and also added extra interactive features and behavior to it, besides the ones introduced merely with the wxFormBuilder.

In-between this process we reloaded the visual models invoking the wxFormBuilder from IDE at compile-time with the staged meta-code we added in the application's source code and regenerated the XRC files many times, to test that no maintenance issues arise by this cycle.

We continue discussing the case study not only regarding the methodological details, but also elaborating on a few important practicing patterns that emerged in the process.

We elaborate on the way composition on user-interface code through ASTs has been applied in the context of our case study. It should be noted that, although at some points it may look like the effect can be also accomplished by typical runtime composition at the level of widgets, in general it is not. In particular, not all widget libraries offer runtime name-based registries for widgets, neither all of them

facilitate the runtime registration of event handlers in the form of typical method invocations.

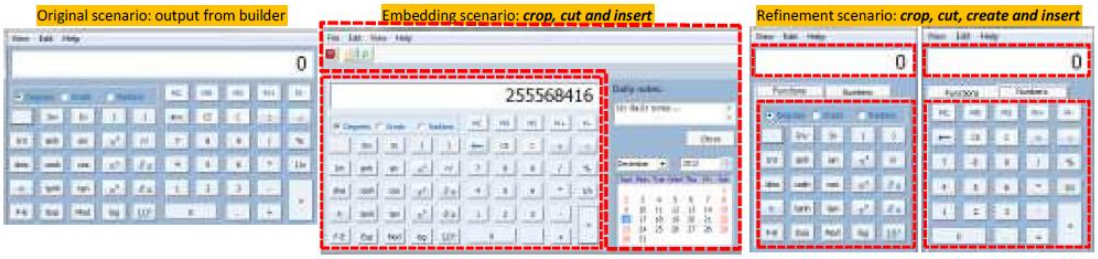


Figure 35. Two example scenarios (middle, right) of user-interface source code composition relying on AST manipulation on top of the original GUI authored with the interface builder (left); updates on the scenarios are automated and are directly remapped on top of the original GUI by simple performing recompilation.

In other words, if linkage is required between interaction objects that are constructed by the generated interface code to custom event handlers provided by the application, then it may the case that the only option is making such code fragments coexist at the same source context.

In our case study, the initial source code corresponding to the outcome of the wxFormBuilder has the following structure (pseudo code, many details removed), and creates the calculator instance shown at the left part of Figure 35:

```
1:  new main frame m_frame0          : null
2:  new panel m_panel0               : m_frame0
3:  new num panel m_panel1           : m_panel0
4:  new num buttons m_button<i>      : m_panel1
5:  new func panel m_panel2          : m_panel0
6:  new func buttons m_button<j>     : m_panel2
```

Figure 36. The GUI parent object typically required

In Figure 36 is depicted the GUI parent object typically required, while line numbering is used only to help in our explanations. Now, we need to perform the following changes: (1) drop the code producing the outer frame (line 1); (2) insert code for event handling implementing calculations on the numeric and function buttons (after lines 4 and 6); (3) crop the numeric and functions panel (lines 3 and 5); and (4) introduce a tab-box were to insert the cropped code fragments for the

calculator numeric and the functions pad. In all these cases we also rely on the automatic relinking of the parent objects offered by the insertion operator, as mentioned earlier.

```
& calc = nil;            ← a global meta-code variable, carrying the entire AST of the user-interface code
& {                      ← an entire block of meta-code begins here
calc = Converter::xrc2ast("calc.xrc");      ← load XRC definitions and convert to respective AST
Tree::Crop(calc, "m_frame0");               ← drop the outer Frame inserted by wx Form Builder      (1)
Tree::Insert(                               ← insert an application event handler for = button
  calc, "m_button54", "EVT_COMMAND_BUTTON_CLICKED",
  "CalcApp::OnEqual"                        ← handler function provided by the application
);                                                                                                  (2)
Tree::Insert(                               ← insert an application event handler for + button
  calc, "m_button60", "EVT_COMMAND_BUTTON_CLICKED",
  "CalcApp::OnAdd"                          ← handler function provided by the application
);
...rest of event handlers are inserted here for the rest of the calculator buttons...
local numbers = Tree::Cut(calc, "m_panel1");  ← cut the code constructing the numeric panel
local funcs   = Tree::Cut(calc, "m_panel2");  ← cut the code constructing the functions panel       (3)
local panel   = Tree::Get(calc, "m_panel0");  ← get the code creating the main calculator panel
local tabBox    = << code here to create the tab box >>;  ← code placed around <<>> is automatically converted to AST
Tree::Insert(panel, tabBox);          ← insert the code for the tab-box after the code of the calculator panel
local numsTab   = << code here to create the numbers tab entry   >>;
local funcsTab  = << code here to create the functions tab entry >>;                                (4)
Tree::Insert(numsTab, numbers);       ← insert the code for the numeric panel after the code of its tab entry
Tree::Insert(funcsTab, funcs);        ← insert the code for the functions panel after the code of its tab entry
}
...any other meta or normal code may be freely placed here ...
!(calc);                              ← inline the entire AST carried by calc at this source location
```

Figure 37. Meta-code to load, manipulate (four labeled steps) and inline the source code for the modified calculator.

The meta-code implementing these four composition steps is outlined under Figure 37, with many details removed for clarity. Also, the actual conversion from XRC to ASTs is cached and is applied only when an internally produced and stored AST file is older than the supplied XRC file. There is code in Figure 37 appearing with a form << some code >>. This is not a conceptual symbolism, but is syntax relating to

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 merely the character string '1+2'. This is useful when one needs to combine in-place an explicitly written source code snippet with other code fragments that are available directly as AST values. In our example, we quasi-quote the source text producing the numeric and function tab entries (middle of step 4 in Figure 37) and compose them via Tree::Insert with the ASTs earlier extracted from the calculator code.

Paint basic

We used the wxFormBuilder once again and we constructed a simple graphics painting application. The latter was actually practiced in alternative ways, such as with single authoring project or alternatively with multiple independent projects (i.e. multiple XRC models). This way we could also assert the compositional flexibility of our proposed approach in combining independently authored interfaces under a single system. To convert XRC to the Delta language ASTs we used the XRC to Delta AST converter we developed, following the proposed approach. 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 merely with the wxFormBuilder. In-between this process we reloaded the visual models invoking the wxFormBuilder from IDE at compile-time with the staged meta-code and we added in the application's source code regenerating the XRC files many times, to test that no maintenance issues arise by this cycle.

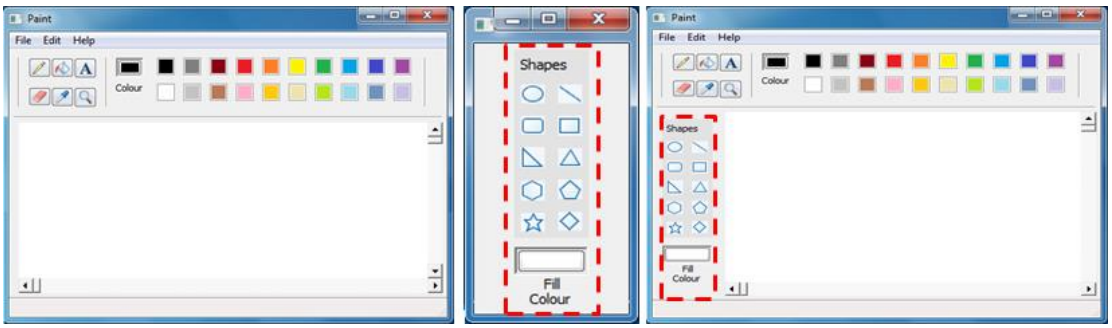


Figure 38. Examples of the generated interfaces: Left: Original application GUI authored by the interface builder; Middle: Custom toolbar authored as a separate interface; Right: Composing the two previous interfaces through AST manipulation.

We used the identified manipulation operators described in above subsection titled as Manipulating User-Interface Code as ASTs and we implemented several composition scenarios. Figure 38 illustrates one of the implemented user-interface composition scenarios based on two separate interface descriptions. The toolbar of the second interface is initially retrieved by cropping its top level frame, and is then

inserted directly in the top level frame of the paint application. Finally, the combined interface is produced by inlining the transformed paint application AST.

Self MDE deployment's dialog

One more opportunity for a case study in User Interface Builder was provided by the development of the self MDE deployment of our approach described in 3.1.1. We developed the simple application dialog which is depicted in Figure 39. We used the wxFormBuilder to construct the XRC model of the dialog and we converted the XRC model to AST with the developed converter. Then, using the metaprogramming features of the Delta language, we imported and manipulated the Model Editing AST, and also added extra interactive features and behavior to it. In particular, we added three events for the buttons and one for the choice of the model in order to launch or edit it.

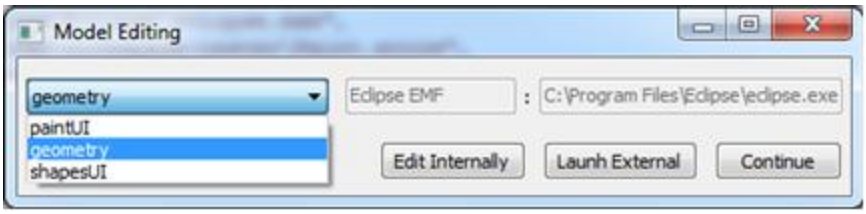


Figure 39. The dialog open at compile-time to handle the models of development

In between this process, using the metaprogramming features of the Delta language, the *wxFormBuilder* was launched directly from the meta-code during compilation to allow interactive editing of the user interface. We repeated loading of the visual model and regenerating of the XRC file numerous times, so as to test that no maintenance issues arise by this cycle.

4.2 Class Builder

In this section we describe the deployment of the approach, focusing in Class Hierarchy. Additionally, to test our approach and assess its expressive power and engineering validity, we have carried out case studies. In particular, we have developed *Geometry* application described in section 4.2.2 and a *Library* application described in section 4.2.3. We continue with the description of the Class-Hierarchy deployment of our approach and then following the case studies.

4.2.1 Applying our approach for Class Hierarchy

Following the proposed approach we deployed it for Class Builder as it is outlined in Figure 40. We used the Eclipse Modeling Framework to model a class hierarchy for the development of an application. The model is created through the Ecore meta-model and its specification is generated in XMI format. Then, to convert XMI to Delta language ASTs we built an appropriate converter we implemented for the demands of this case study that parses the XMI data and maps the model entities to corresponding Delta code structures.

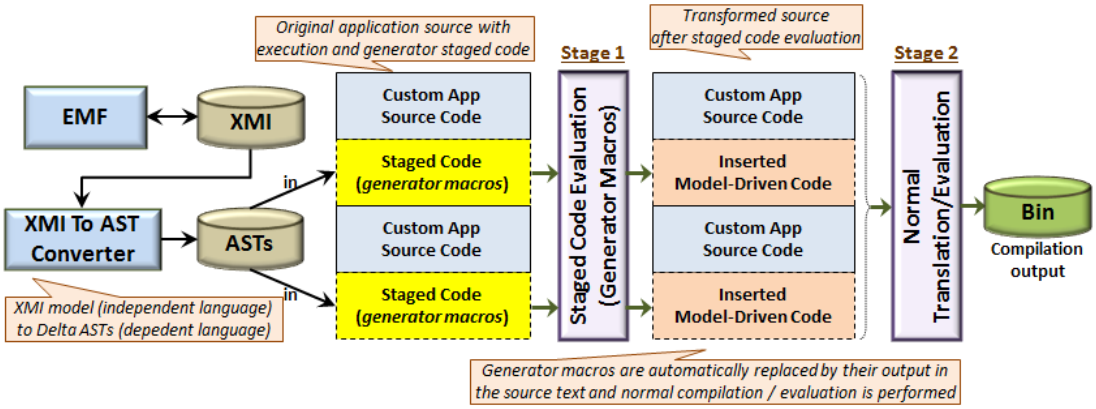


Figure 40. Deployment of approach focusing on Class builder

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.

Additionally, for this scenario, the MDE tool deployment was implemented using two different approaches. The first one again involved launching an external tool to update the model, in this case the Eclipse Ecore model editor. The second one focused on implementing the model editor as an inherent part of the metaprogram, i.e. without launching any external applications. Of course, such a custom editor need not be implemented from scratch but may reuse any model editing library implemented in the same language.

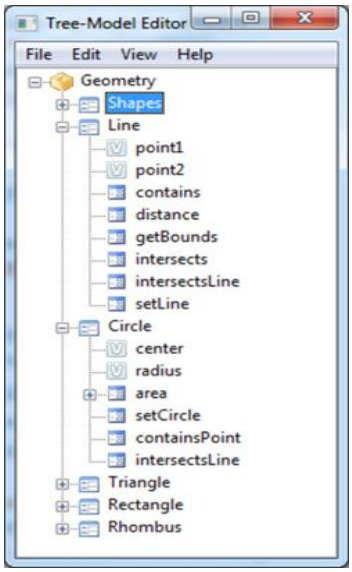


Figure 41. The internal custom model editor launched during compilation case study of Geometry.

Towards this direction, we implemented a simple GUI (see Figure 40) offering an editable tree control to specify the class hierarchy, effectively emulating the Ecore model editor functionality. Using this approach we may take advantage of executing in the same address space and also store the generated ASTs in a metaprogram variable that can be later used directly in the generator macros, thus removing the need for reloading the AST data from storage.

Figure 42 illustrates the metaprogram used to implement the second case, while for the first case the deployment code closely resembles that shown in Figure 29, but with the invocation of the MDE tool targeting the Eclipse model editor.

```
&function LaunchEditor(modelPath){  ←a meta-function available during staged evaluation; can be put in a library
  local frame = wx::frame_construct(nil, "ID_ANY", "Tree-Model Editor"); ← create GUI frame
  local tree = wx::treectrl_construct(frame, "ID_ANY", "tree"); ←create model editor tree control
  local model = load_xmi(modelPath);        ← load the XMI model definition and store to meta-code variable
  PopulateTreeFromModelData(tree, model);
  function OnSave() {                       ← event handler for saving the updated model
    local model = ExtractModelFromTree(tree);
    xml::store(model, modelPath);           ← store the updated model at the same path that we loaded it from
  }
  frame.connect("ID_SAVE", OnSave);         ← register events handlers for menu items and tree control actions
  frame.connect("tree", "EVT_COMMAND_TREE_ITEM_ACTIVATED", OnItemActivated);
  ...any other event handlers are registered here...
  frame.show();
  wx::main_loop();       ← execution is passed to the GUI and will resume only when the editor is closed
}                        ← this means that during model editing staged evaluation is practically stalled
&const modelPath = "./models/Geometry.ecore";       ← a meta-code variable holding the path to the model
&LaunchEditor(modelPath);     ← function call executes during stage evaluation (executor macro) launching the editor
&model = load_xmi(modelPath);          ← load the updated XMI model definition and store to meta-code variable
&ast = Convert::xmi2ast(model);        ← convert the model to the respective AST
&ApplyModelExtensions(ast);  ←optionally, extend or update the generated model code as needed through AST editing
...additional normal user code that has to be placed before the generated model code can be placed here...
!(ast);                                ← inline the entire AST value at this source location (generator macro)
...additional normal user code that has to be placed after the generated model code can be placed here...
```

Figure 42. The internal editor code as an inherent part of the staged metaprogram.

4.2.2 Developing Applications

Geometry

In the beginning, we used the Eclipse Modeling Framework to model a class hierarchy for the development of a simple Geometry. The hierarchy contained the abstract notion of shapes, as well as concrete drawable shapes like points, lines, circles, etc. The model was created through the Ecore meta-model and its specification was generated in XMI format. Then, we converted the XMI model to Delta language ASTs using the appropriate converter we built.

In the Figure 43 is depicted the model, the generated code structure (shown as code, but is in fact 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 43 top-left, highlighted). The second one did not involve any model editing, but relied on obtaining the generated AST and inserting the method bodies directly into it as part of the staged code evaluation (Figure 43 bottom, 2nd statement). This approach may seem more difficult to adopt, but in fact it is easy to develop and offers several advantages over the first one.

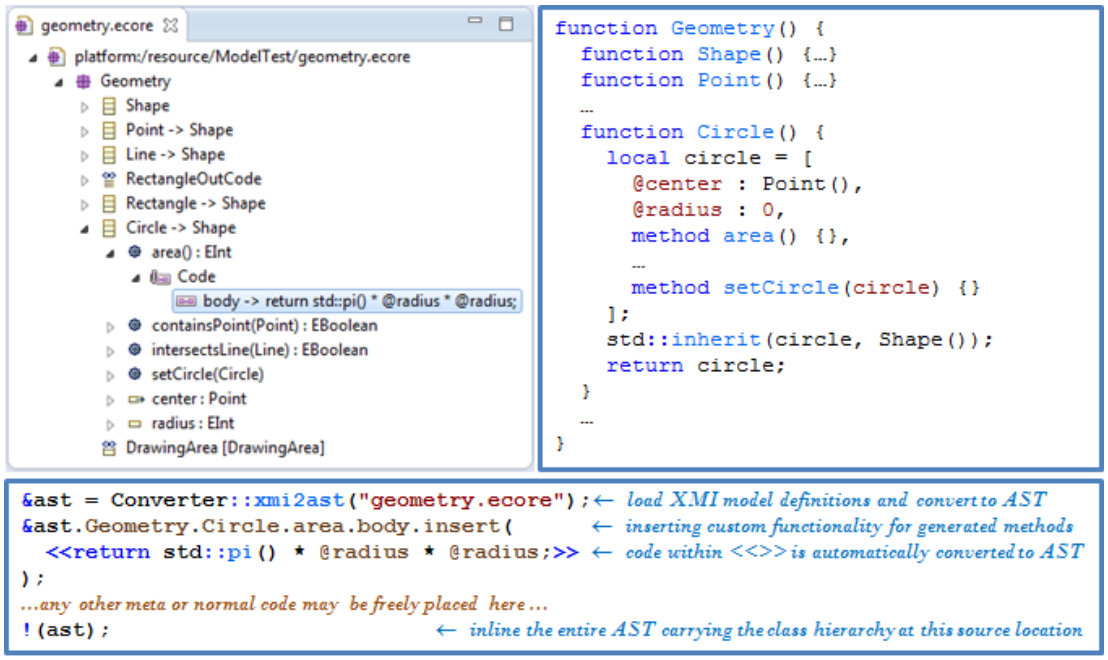


Figure 43. Top-left: Ecore model of the target class hierarchy; Top-right: Code structure (AST) generated by the model; Bottom: Deployment code for loading and converting the model to AST, performing manual updates through AST editing and inlining the final AST code. The initial value of the meta-variable ast corresponds to the code structure shown at top-right.

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 then use their 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.

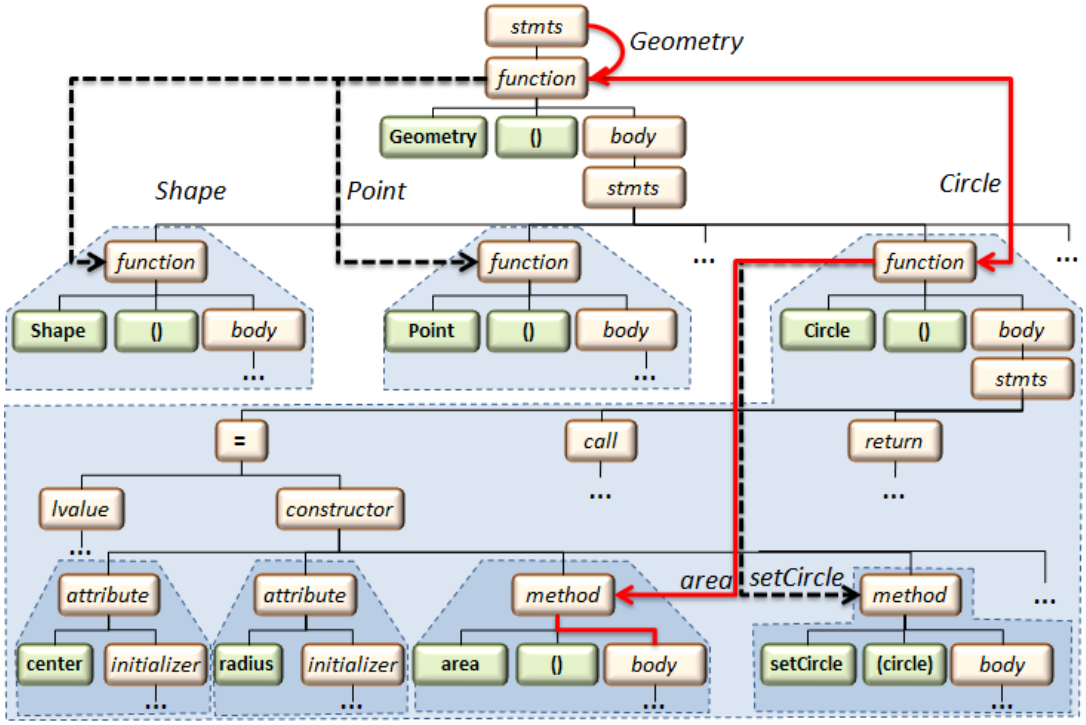


Figure 44. Supporting quick access to all class hierarchy entities through AST decoration. The AST shown corresponds to the generated hierarchy of **Figure 43**, while the highlighted path `ast.Geometry.Circle.area.body` was used to insert custom method functionality.

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, modular composition, etc. The main issue related to programmatically extending the originally generated AST is that we need to traverse the AST to locate the nodes to be extended and therefore requires knowledge of the code generation scheme

utilized to form the particular AST structure as well as internal AST information. To relieve the developer from having to know such details, we utilized an AST decoration process to allow direct navigation across AST nodes using the named entities of the class hierarchy (see Figure 44). This way, knowledge of the model entities and a simple tree manipulation API are sufficient for a developer to introduce elaborate AST extensions.

During the development of this Case study, we deployed the compile-time invocation of MDE tools in two different ways as we discussed in the previous section in order to examine the convenience and effectiveness of this type of launch an MDE tool during compilation.

Library basic

Once again, we used the Eclipse Modeling Framework to model a class hierarchy for the development of a simple Library application. The hierarchy involved the notion of a library in which Books (EClass), bookType (EEnum), Dictionaries (EClass), Magazines (EClass), Writers (EClass), PublishingHouses (EClass) etc are included. The model was created through the Ecore meta-model and its specification was generated in XMI format. Then, we converted the XMI model to Delta language ASTs using the appropriate converter we built. The call of this converter is depicted in label 2 of Figure 45.

For the method implementations of the modeled classes we used the second method of the previous two discussed in the case of *Geometry*, which relies on obtaining the generated AST and inserting the method bodies directly into it as part of the staged code evaluation as depicted in labels 3, 4 of Figure 45.

Then, we developed pure functionality for the library application, with details removed for clarity. We created instances of writers, books and then an instance of a library with a list of books as outlined in label 5 of Figure 45. Finally, we searched for the books published in 2009 and they are in the Library as depicted in label 6 of Figure 45.

```
using wx;                    ← normal code, directive for importing the wxWidgets GUI toolkit

& libClasses = nil;          ← a meta-code variable, that will store the AST of the library application class hierarchy code
& {                          ← an entire block of meta-code begins here                                                    (1)
libClasses = Converter::xmi2ast("lib.ecore");  ← load XMI model definition and convert to AST                              (2)

local method_setBook = <<
            method setBook (title, bookType,…) {
                        @title = title;        ← @title is one attribute of the class Book
                        @type  = bookType;     ← @type is one other attribute of the class Book
                        …                                                                                                  (3)
            }
>>; ← code placed around<<>> is automatically converted to AST
…other asts of methods or bodies code may be freely there…

libClasses.library.Book.type.push_back(method_setBook);  ← add code method in Book class of the
                                                         ← library's AST                                                   (4)
…other additions–transformations of library ast may be freely there…

}                                        ← the block of meta-code ends here                                                (5)
…any other meta or normal code may be freely placed here…
!(libClasses);                           ← inline the entire AST carried by libClasses at this source location
…any other meta or normal code may be freely placed here…
book1 = library().Book();                                  ← create a default book instance
Book1.setBook("Inferno", "mystery thriller", 2013);        ← set data to the book instance
…any other meta or normal code may be freely placed here…
Library = library().Library();                             ← create a default library instance
Library.name = "library UOC";                              ← set name of the library instance
Library.books.push_back(book1);                            ← add book1 in the list of library's books
…any other meta or normal code may be freely placed here…
foreach (book, Library.books)                              ← print all the books that are published in 2009
            if(book.year == 2009)                                                                                          (6)
                        std::print(book.title, "by", book.writer, "\n");
…any other meta or normal code may be freely placed here…
```

Figure 45. Meta-code to load, manipulate and inline the source code for the library.

Again during the development process of the case study, we reloaded the model and regenerated the XMI specification to verify that no maintenance issues were introduced in the development process. Additionally, we added staged code in order to invoke the MDE tool at compile-time with the correspondent XMI model in order to launch it.

4.3 Automatic User Interfaces

An alternative way to construct User-Interfaces instead of using a WYSIWYG tool is the automatic generation source code by specification. In some way, the latter is the model which describes the User-Interface. The automatic UI generation tool gets this specification as input and either creates the UI and runs the application, i.e. in the

case of executors or creates the correspondent UI source code of the application and then source code is manually completed in order to finalize the system under study. The latter is the kind that we focus on in this thesis and it also causes the maintenance problem similar to the WYSIWYG tools we discussed previously.

In this case study, we developed a system which gets a specification of annotated APIs as input and delivers an AST according to our approach, instead of source code. The goal of this case study was dual: (a) to research an alternative way for Model-Driven Engineering of User-Interface; and (b) to deploy it in our approach. We continue with the description of the specification with annotated APIs of User-Interfaces. Then, we briefly analyze the system which produces the AST. Afterwards, we describe a pure user interface of a library application we built using this approach.

4.3.1 Defining an alternative UI model

In User-Interfaces, it is common to define models using MDE tools and save them in the form of User Interface Description Languages (UIDL). An alternative model that can be defined, is the specification annotated APIs. We have adopted the annotated user interface APIs based on lectures of the computer science department of the University of Crete, Development of Intelligent User-Interfaces and Games [45]. In general, when the construction of a software system begins, the operations that the system will eventually support are defined. So, a UI model could be a specification which includes the operations of the system under construction. In our case the Specification (model UI) is defined by ***Operation***, **Signature**, **Parameter**, ***returnValue***, ***func***, ***dataFlowType***, ***typeInfo***. Each of these model's constructs are described below.

Operation

One *Operation* can be defined as a User Interface or a non-User Interface operation. Each operation consists of the *signature* and the *func*.

Signature

The operation's signature consists of the *name*, the parameters and the *returnValue* which gets a value when the reference function (*func*) is fired by the user. The *name* of the operation's signature must be unique in a defined specification.

Parameter

Each parameter consists of the *name*, the *dataFlowType* and the *typeInfo*. It models the constituents of one operation. The parameter's name must be unique in the operation.

returnValue

It refers to the result of the function call of the operation's ref (*func*). It contains the *name* and the *typeInfo* of the return value. When the application starts *returnValue*'s value is empty.

func

The *func* includes the *ref*s of the operation's functions which the user defines for the system's operation. These functions are callback and are fired during use of the application.

dataFlowType

The dataFlowType is the type of parameter or result data. The possible values are *"In"*, *"Out"* and *"InOut"*. *"In"* refers to data passed to the Operation's callback function (*func*). *"Out"* refers to data coming from the Operation's callback function (*func*). *"InOut"* refers to data passed and coming from the Operation's callback function (*func*).

typeInfo

It describes the type of operation's *parameter* or *returnValue*. It can be a basic type (*String*, *Boolean*, *Integer* and *Void*) or defined type that is described by a Table. For example a String is described by this table `[ @type : "String" ]`. The defined types are described by a Table, but the contents vary. The available defined types are *Struct*, *List*, *Vector*, *Array*, *Enumeration* and *Union*. These types are described by the following tables:

Struct

```
[
        @type : "Struct",
        @userDefClassId : <String>,
        @members :
        [
                [
                        @name : <String>,
                        @typeInfo : <basic type> | <user-defined type>
                ],
                …
        ],
]
```

List/Vector

```
[
        @type : "List"/"Vector",
        @userDefClassId : <String>,
        @elementTypeInfo : <basic type> | <user-defined type>
]
```

Array

```
[
        @type : "Array",
        @userDefClassId : <String>,
        @elementTypeInfo : <basic type> | <user-defined type>
        @length : <Number>
]
```

Enumeration

```
[
        @type : "Enumeration",
        @userDefClassId : <String>,
        @members :
        [
                [ @name : <String>],
                …
        ]
]
```

Union

```
[
        @type : "Union",
        @userDefClassId : <String>,
        @members :
        [
                [
                        @name : <String>,
                        @typeInfo : <basic type> | <user-defined type>
                ],
                ...
        ]
]
```

Each of the types defined above can be described with correspondent User-Interfaces. An example of the basic types' String correspondent User-Interface is depicted in Figure 46. An example of the description of the List type is depicted in Figure 45.

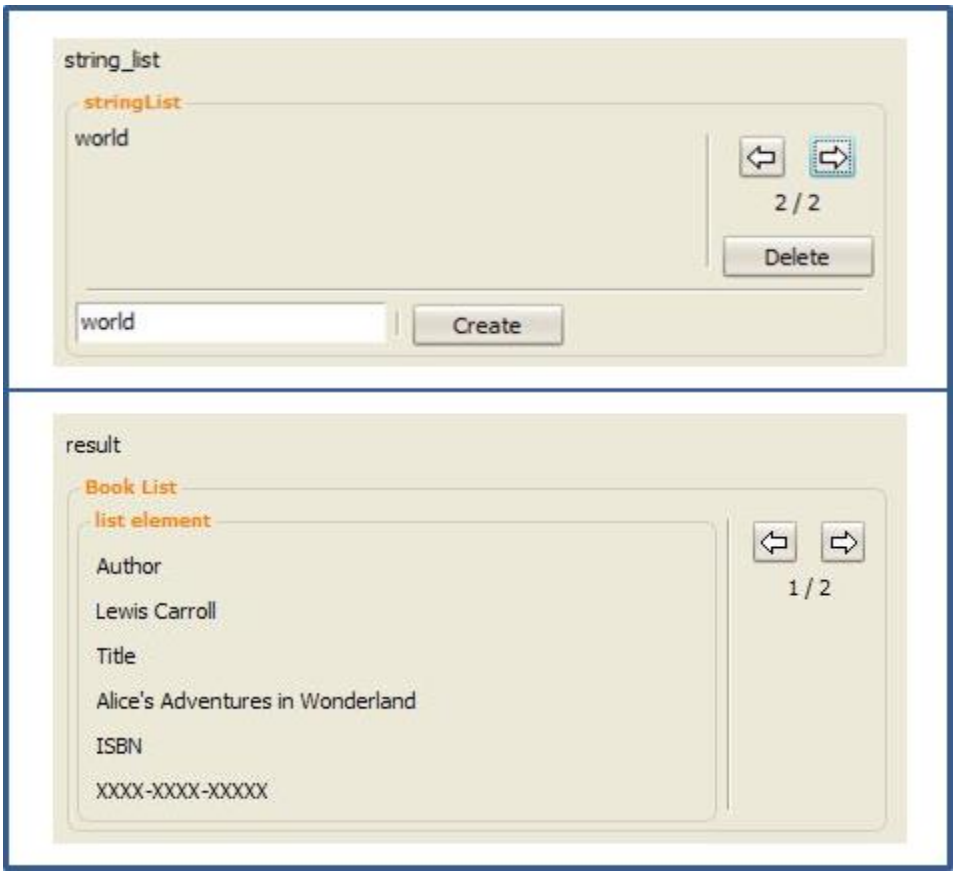


Figure 46. List's User Interface description; at the top there is the description when dataFlowType is "In" or "InOut"; at the bottom there is the description when dataFlowType is "Out".

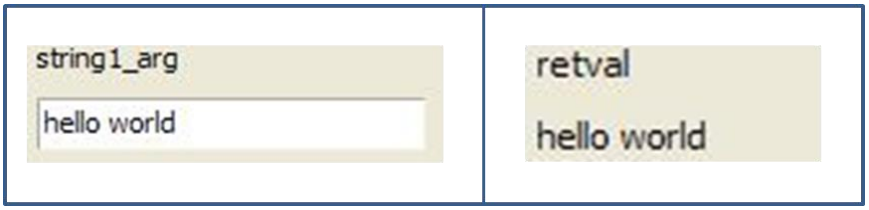


Figure 47. String's User-Interface description; on the left side is the UI when *dataFlowType* is "In" or "InOut"; on the right side is the UI when dataFlowType is "Out".

A definition example of a calculator operation is,

```
calcFuncSpec = [
    @signature : [
        @name : "calc",
        @returnValue : [
            @name : "result",
            @typeInfo : intTypeInfo
        ],
        @parameters : [
            [ @name         : "operand_1",
              @typeInfo     : intTypeInfo,
              @dataFlowType : "In" ],
            [ @name         : "operand_2",
              @typeInfo     : intTypeInfo,
              @dataFlowType : "In" ]
        ]
    ],
    @func : [
        @ref : "UserFuncsEvt::calcFunc"
    ]
];
```

The specification (model UI) is defined in a script source file in which there is a function *GetAPISpec* which returns the API specification object.

4.3.2 The Auto-generation UIAPI engine

In order to create the User-Interface and the API from the UIAPI specification model which is described above, we developed an appropriate engine. This engine gets the UIAPI specification model as input and translates it according to the theory of lectures of the University of Crete 'Development of Intelligent User Interfaces and

Games' lesson [45] in the correspondent Delta Language AST instead of source code in respect to our approach.

The *Auto-generation UI engine* is composed as depicted in Figure 48 by the following main parts; the *UIAPISpecValidator*, the *UIBuilderCore*, the *RulesMap* and the *MicroUIsBuilder* which are briefly described below.

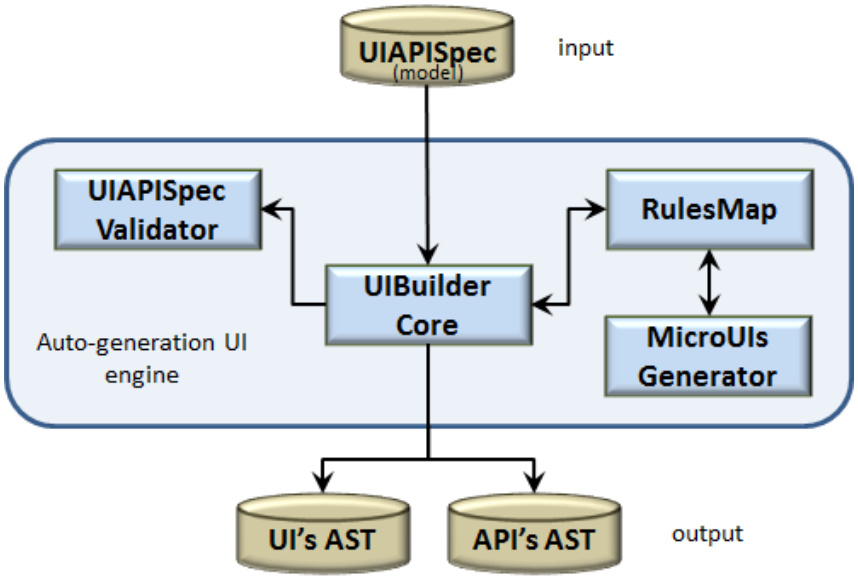


Figure 48. The auto-generation UI engine architecture

The ***UIAPISpecValidator*** is the first step of the engine. It handles the validation check of the model given as input to ensure it was appropriate. In case the model is not valid, the engine gives the appropriate error message for the model and terminates.

The ***UIBuilderCore*** is the main part (core) of the engine. It constructs the main frame's User-Interface and then uses the operations described in the specification (model) in order to construct their UI and API. Finally, it returns the object with the ASTs result of the engine.

The ***RulesMap*** is also an important part of the engine which is responsible for the theory of lesson lecture [45] in order to create the UI for the parameters and the return value of the operation.

The ***MicroUIsBuilder*** is the builder for each of the types that could be defined in the specification. It is called by the *RulesMap* in order to create parts or whole UIs of the parameters and the return value.

4.3.3 User-Interface Design Issues

As it can easily be perceived, the User-Interface created by this auto-generation tool is predefined for each of the UIs and its design is sometimes far from the desired design of the application graphical user-Interface. The accuracy of the User-Interface design for an application is very important, so we have to address this issue in this case. There are two approaches to solving this problem without causing troubles in the maintenance issue we address.

The first way to fix the design issue is to enrich the specification model in order to define the design of the User-Interface of the application under study. In the one hand, the new specification model demands the extension of the auto-generation UI API engine in order to create the correspondent UI source code for each model. On the other hand, the more enriched the specification model is the more difficult it will be to define the specification script correctly. In order to define this type of models effectively, an appropriate visualization software tool will be needed in which the user will create the model (script) automatically. This work has not been done in the thesis and it is subject to future work.

The second way to address the issue of the design of the GUI is to get the current result of the auto-generation engine UI and API AST. Then, using the *Manipulating Interface Code as ASTs* which we discussed previously, we could transform the produced AST in order to edit the User-Interface of the application under study. In the one hand, this solves the problem of the design accuracy of the User-Interface although one or more changes to the API's AST will be needed because of the dependencies between the two auto generated ASTs (UI & API) from the engine. The API's AST transformations need to be done in case of UI changes like replacing a

widget and are not needed in case of setting data in a widget or changing its position in the application frame. This means that although the double transformations of the UI AST and API AST are not so easy, the cases in which they are really needed are infinitesimal. So this is the approach we use in the following application of a Library in order to edit its User-Interface.

```
&using #UIBuilder;
&using #UIAPIModel;

&ast = nil;
&{
  ast = UIBuilder::AutoGenUIAPI (UIAPIModel);
  …other additions –transformations of library ast may be freely there…
}
 …any other meta or normal code may  be freely placed  here…
!(ast.gui);
 …any other meta or normal code may  be freely placed  here…
!(ast.api);
     ……
```

Figure 49. Default embedded metacode using the auto-generation tool we developed

In order to use the auto-generation tool need to write the metacode as it is outlined in Figure 49. Developers have to include the *UIBuilder* as embedded metaprogram in the program under study in order to use the auto-generation engine. Additionally, they have to include the specification script (model). Then developers include a call to the auto-generation engine as an embedded metaprogram in order to create the ASTs. Afterwards, they write the transformations for the User-Interface and the API ASTs. This MDE process is the same as previously described. The only change is the way of construction of the model in which the user has to write in a script in order to construct it. We can easily try the self-deployment of the Specification and change it at compile time before the evaluation of the specification. In this case though there is no point in intervening because the model is a script in the form of source code and can be edited from IDE at development time. The self-deployment could be done in case we had built the visualization tool we discussed above.

4.3.4 Developing UI for a Library

To test this approach and examine the effectiveness and engineering validity we have carried out a case study. We developed a pure library application. We defined four operations for the library. The view, search, rent and let a book operation of a library. We created the model defining four different operations: ***viewBooksOfCategory***, ***searchABook***, ***rentABook***, ***returnArentedBook***.

- The ***viewBooksOfCategory*** describes the operation in which users choose the category of a book, and can view the books there are in Library. It is a UI operation in the specification model.
- The ***searchABook*** describes the operation in which the user searches for a specific book giving title/writer's name/year/type of the book. It is a UI operation in the specification model.
- The ***rentAbook*** describes the operation in which the user rents book(s) from the library
- The **returnArentedBook** describes the operation in which the user returns to the library rented book(s).

All these operations are UI operations and none of them are non-UI operations. For each of them, we described the parameters, the return value and the *ref* function operation.

Then, we converted the specification to ASTs using the auto-generation engine we developed. We get two ASTs as result; the first is the API and second is the User-Interface of the application as it is depicted in the Figure 51.

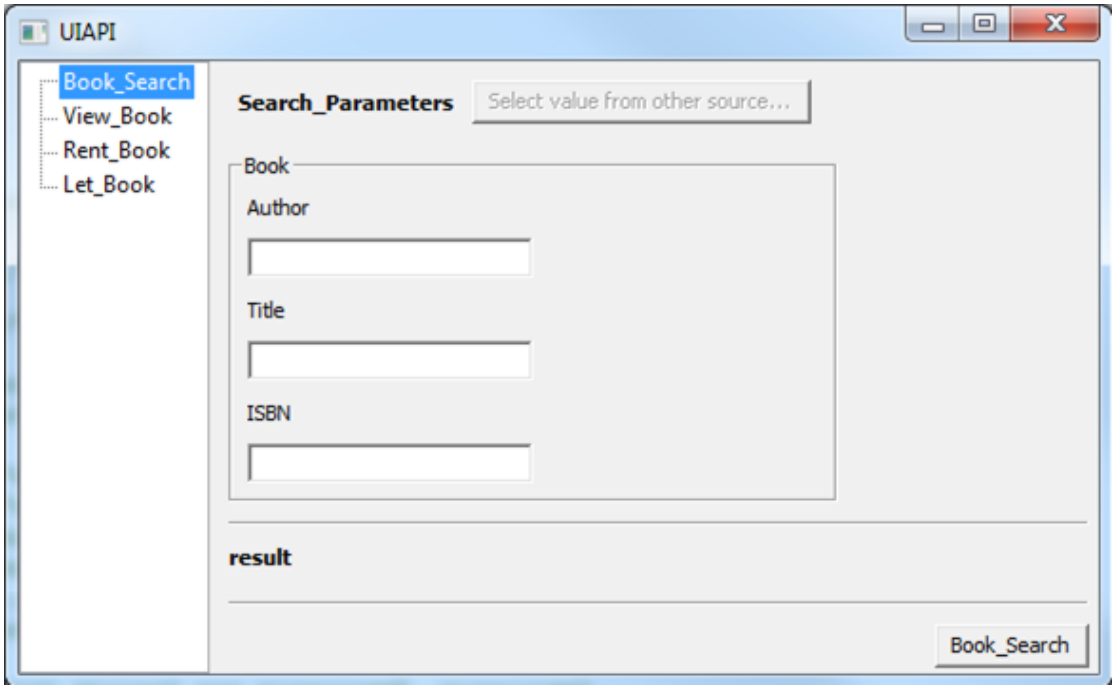


Figure 50. User-Interface produced by auto-generation UI engine.

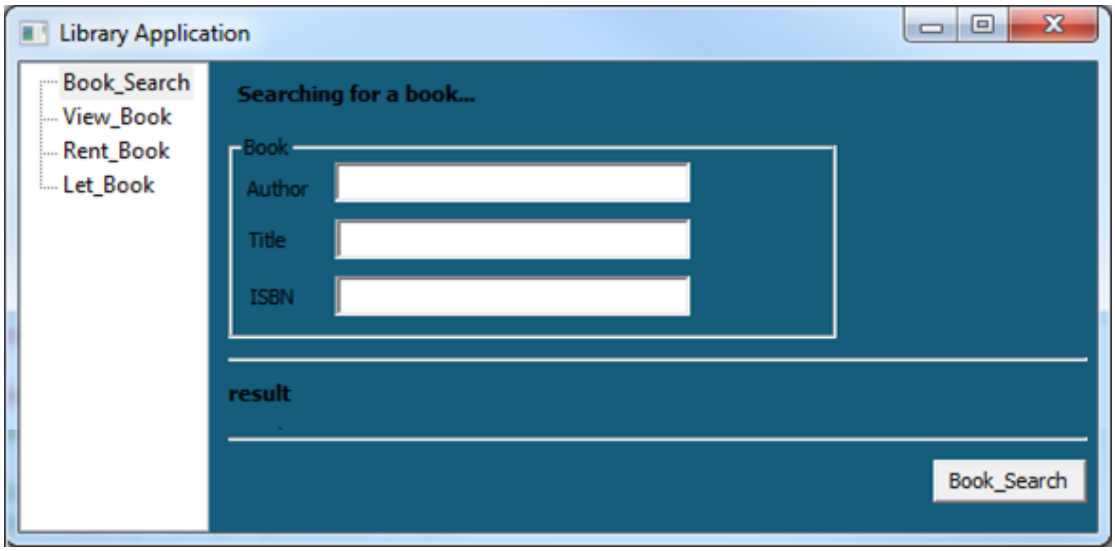


Figure 51. Manipulating the auto-generated User-Interface through ASTs transformations.

Afterwards, we edit the default auto-generated AST of User-Interface using the Manipulating User-Interface Code as ASTs operators we described in previous section as it is depicted in the Figure 51 in order to adapt the auto-generated UI for the Library application. The meta-code implementing the transformations for the UI of the *Book_Search* is outlined in the Figure 52, with many details removed for clarity. Firstly, we include the specification of library (UI model), the UIAPI engine and the

library of manipulating User-Interface operators (see step 1 of Figure 52). Then, we changed the default title “UIAPI” with title “Library Application” and the background color of the frame (see step 3 of Figure 52). Then, we removed the inactivated button which is unnecessary in current UI and changed the *textctrl* titled “Search_Parameters” with more specific title “Searching for a book…” (see step 4 of Figure 52) and the positions of the *textboxes* in the sizers (see step 5 of Figure 52). Afterwards, we inline the ASTs between the custom source code of the Library application (see step 6 of Figure 52). Finally, we complete the reference functions of the operations by pure source code to add their functionality.

```
using wx;                    ← normal code, directive for importing the wxWidgets GUI toolkit

& using #libSpec;            ← meta directive for importing the model UI specification of Library
& using #UIAPIEngine;        ← meta directive for importing the UI API Engine                          (1)
& using #UIAST;              ← meta directive for importing the library script of the manipulating UI ASTs operators

& asts  = UIAPIEngine::CreateUIAPI(libSpec);   ← call the UIAPIEngine to create the ASTs              (2)

& {                          ← an entire block of meta-code begins here
// gui ast transformation to change frame's title and bgcolour
UIAST::addMethodcall(asts.gui, "UIFrame", "settitle", "Library Application");                          (3)
UIAST::addMethodcall(
            asts.gui, "UIFrame","setbackgroundcolour", <<wx::colour_construct(22, 93, 125)>>);
UIAST::addMethodcall(asts.gui, "Book_Search", "settitle", "Searching for a book…");
// gui ast transformation to remove useless (in our case) button                                       (4)
UIAST::remove (asts.gui, "Book_Search_button_selectvaluefromothersource");
// gui ast transformation to change position of textctrl widget of Author
local sitem1 = UIAST::getchild_index(asts.gui, "Book_Author_textctrl");
UIAST::remove(asts.gui, "Book_Author_textctrl");                                                       (5)
UIAST::insert_after(asts.gui, "Book_Author_statictext", sitem1);
…other meta code for UI transformations is inserted here…
}                                ← the block of meta-code ends here
…any other meta or normal code may be freely placed here…
!(asts.gui);                     ← inline the entire AST for the GUI of the library application        (6)
…any other meta or normal code may be freely placed here…
!(asts.api);                     ← inline the entire AST for the API of the library application
…any other meta or normal code may be freely placed here…
```

Figure 52. Meta-code to include the specification(model UI), the UIAPI engine and the library of manipulating UI for AST’s operators in label 1; Meta-code to call the auto UIAPI engine in label 2; Meta-code to transform the auto-generated AST’s GUI in labels 3,4,5 and inline the ASTs in order to generate the Library’s application source code in label 6.

4.4 Combined Deployment

In general, during the development process it is common to use more than one model-driven (MDE) tool to construct a single application. Each MDE tool is used to construct one or more models. As well in our approach, we can use one or more MDE tools. For each of the XML models that constitute the deliverables of model-driven tools will be converted in ASTs by appropriate converters (we have to build a converter for each different modeling language). Then, developers have to handle the produced ASTs once by transforming them and inlining ASTs or parts of them in appropriate positions of source code according to general approach for code manipulation and insertion using ASTs is the one earlier described in section *Multistage Languages* and relates to compile-time metaprogramming languages, involving two stages that are also depicted under Figure 53: meta-code evaluation (stage 1), and normal compilation (stage 2).

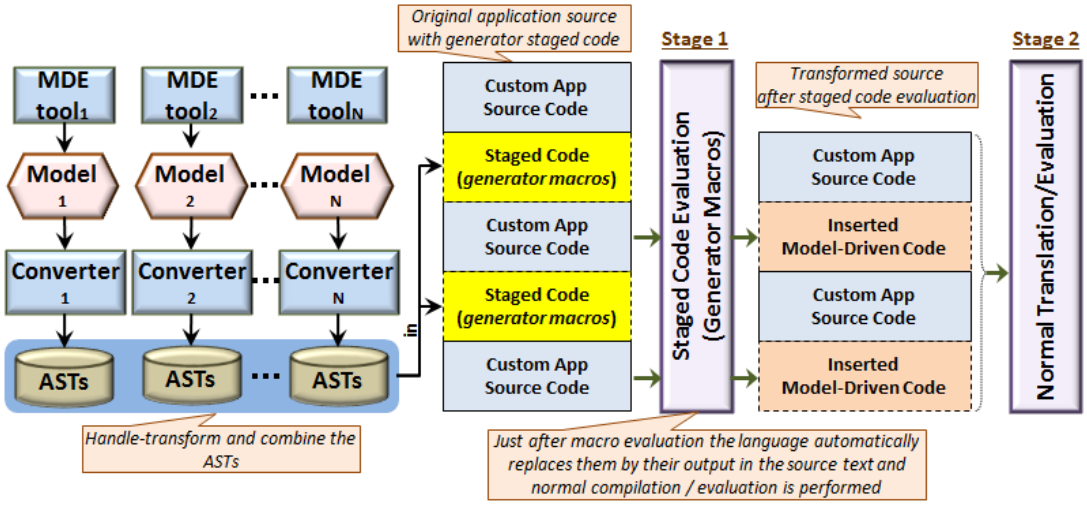


Figure 53. Deployment of approach focusing on combined deployment; use more than one MDE tools.

We continue with the presentation of the case studies, using more than one MDE tools for a single application. In particular, we have carried out two case studies. The first case study is the Paint in which we used the wxFormBuilder and the EMF tool. In the second case study we developed a Library application in which we used the EMF tool and the Automatic User-Interfaces we described in section 4.3.

4.4.1 Developing a paint application

Using the wxFormBuilder we constructed a simple Paint application. The XRC model which delivered by wxFormBuilder is the first of three models built for development of Paint. Then, using again the wxFormBuilder we constructed one toolbar as it is depicted in the Figure 38. Finally, we use the model delivered for shapes class hierarchy of previous case study. The latter was changed and extended to be appropriate for Paint application.

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. It is important to note, that although a simple concatenation of the generated sources caused no direct compilation conflicts, it was far from sufficient for deriving a fully-functional application.

In fact, multiple manual updates were necessary involving both generated components and requiring bidirectional dependencies. Firstly, the event handling code required knowledge of the separately generated implementation classes. 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 54, with details removed for clarity.

Initially, the XRC interface definitions for both the basic paint application UI and the shapes toolbar extension are loaded and converted to AST. Similarly, the XMI model definition for the shape toolset class hierarchy is also loaded and converted to AST (step 1). Actually, all such ASTs are cached and the conversion is only applied when the internally produced and stored AST file is older than the supplied model file.

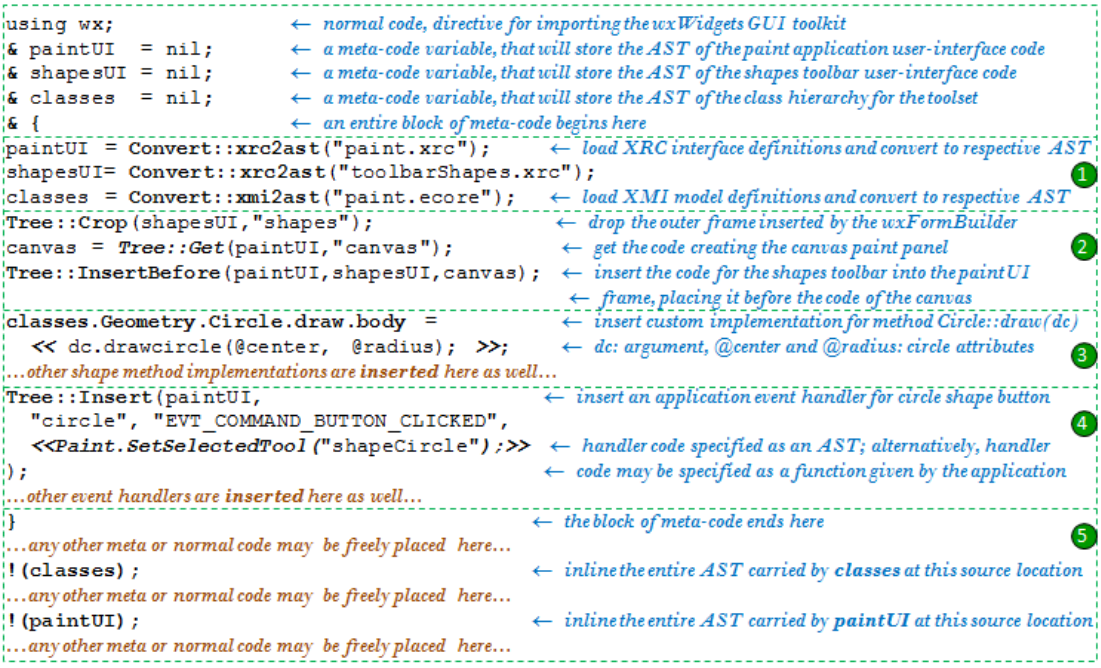


Figure 54. Meta-code to load, manipulate and inline the source code of all modeled aspects of our system. The result is a fully functional paint application like that shown on the right of Figure 38.

Then, the interface definitions are combined 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. With the visual representation ready, the next step involves implementing the various methods of the class hierarchy (step 3). This is achieved by creating and inserting AST values in the method bodies as discussed in the previous section. Notice that the quasi-quoted code can directly link to UI elements. The next step is the generation of the event handling code (step 4). As shown, we can specify event handling code directly as an AST, while the code itself may refer to objects related to the shape toolset class hierarchy. 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 5). 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.

Specifically for the user-interface code, it should be noted that it may have been possible to accomplish the same result using typical runtime composition at the level of widgets. However, such an approach cannot be deployed in general, as there are widget libraries that offer no support for name-based registries for widgets, or runtime registration of event handlers in the form of typical method invocations. In such cases, if an object constructed by the generated interface code needs to be linked to custom event handlers provided by the application, then making such code fragments coexist at the same source context may sometimes be the only solution.

4.4.2 Developing a library application

In this case study, we used the EMF tool and the auto-generation which is described in section 4.3. Our purpose for this case study is to examine whether the combination of more than one model, although constructed in a different way (specification by writing source code in a script and Ecore by using the EMF in Eclipse), that it does not affect the maintenance issue. In particular we used two of the models we constructed for the previous case studies. Firstly, the EMF model we built in the Library basic (see Figure 55). We changed the pure source code of the Library object creation from the previous case study by adding source code which parses xml file in order to load the Library data and source code to save back the Library data to xml file. In other words, we built a simple database for Library application.

Then, we used the specification (model) which was defined in the Automatic User Interfaces in Library UI example. We replaced the functionality source code of the reference functions of the operations we developed in the previous case study with source code which uses the functionality we developed in the part previously described *above* for the class hierarchy. Again during the process, we reloaded the models and regenerated the XMI specification and the UIAPI specification to verify that no maintenance issues were introduced in the development process.

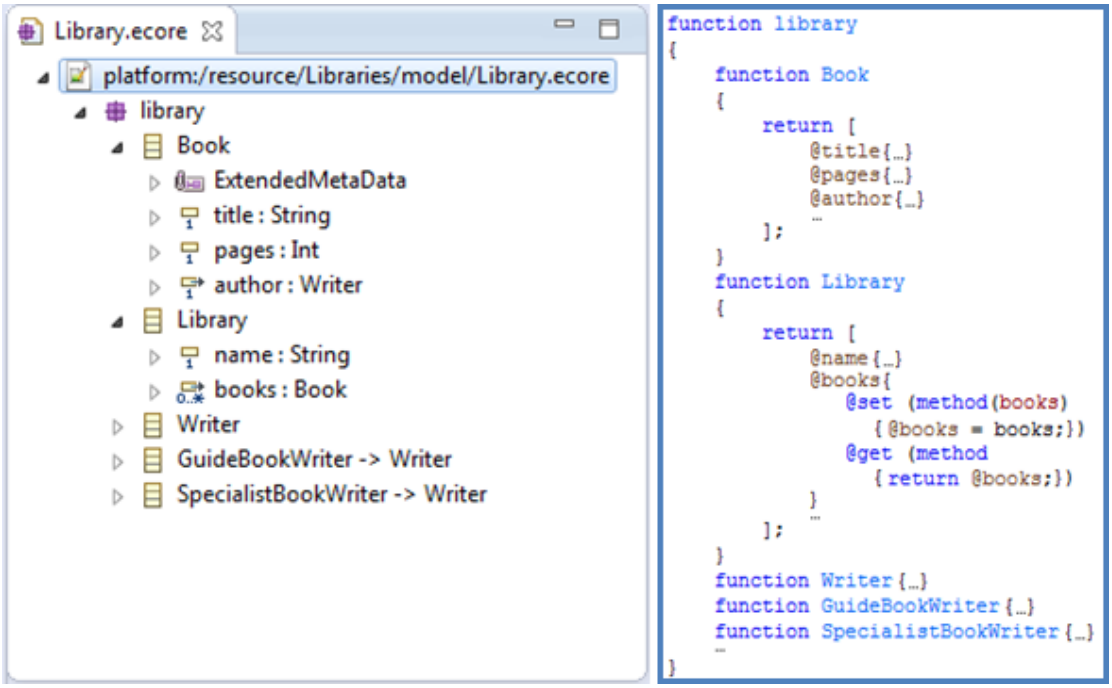


Figure 55. Left: Ecore model of the target class hierarchy; Right: Code structure (AST) generated by the model

Chapter 5

Discussion

In this chapter, we further analyze the problem of maintenance issues by giving a simple example and using traditional generative MDE tools as well as our approach in order to compare them. Then, we discuss the tradeoffs using our approach and finally we describe the applicability of our approach in programming languages.

5.1 Maintenance

We designed a model of the Person class by using a modeling tool which does not deal with maintenance issues. Person includes the attribute *"name"* and the method *"naming"* which sets name in Person. We generate the source code from the model using a code generator. Then, in order to complete the implementation source code of Person we complete the body of method naming. In case we decide to extend or edit the model later, we have to re-generate the source code from the model. The manually written source code in method naming will be lost so we have to re-complete it. In this simple case we just have to copy this fragment of source code before regeneration and then paste it in the updated source code. In a real application development, we design many classes. So, we have to keep old sources'

version and after the regeneration of the source code from the model we have to place the source fragments in the generated source code. This is a very tedious and inefficient process which can cause a lot of issues (e.g. wrong mapping of source fragments in the auto-generated source code).

5.1.1 Addressing maintenance issues so far

In general, the attempts are distinguished in two approaches. The first approach is with the use of annotations within the source code and the second is the support of the full cycle development. We continue describing these two approaches through the example we discussed previously.

Using annotations

Using the aforementioned example, we model the Person class with the attribute *"name"* and the method *"naming"*. Then we create the appropriate code generator (i.e. *.genmodel) from the eclipse modeling framework in order to generate the source code from the model. The source code includes annotations in its comments as it is depicted in Figure 56.

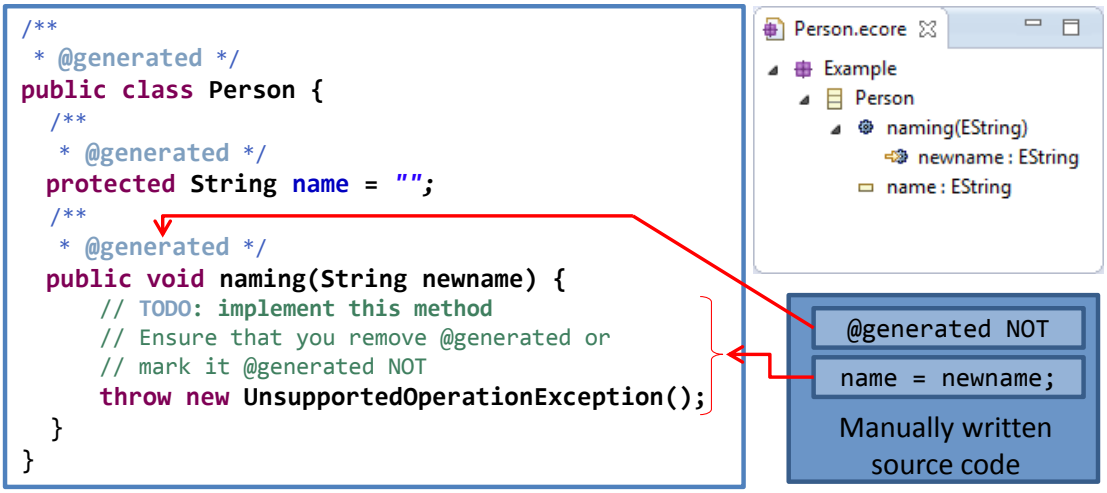


Figure 56. Using EMF tool to design and implement class Person.

Then, we complete the method *"naming"* and replace the annotation `@generated` with the annotation `@generated NOT`. In case we would like to extend the Person model and regenerate the source code, the manually written source code will not be replaced. In particular, code generator parse the generated file and look for the annotations `@generated NOT` in order not to re-create these fragments. If we put aside the additional developers' tedious responsibility of choosing which source fragments to re-generate or not by using annotations, the maintenance issues seem to be solved. In case we consider a different model update involving modifications for already implemented functions, when for example changing the called method *"naming"* with the name *"setName," the* generator parses the file again. This time, the file does not include the method *"setName"* and the generator cannot map this method with the previously generated method *"naming".* However there is no knowledge that *"naming"* and *"setName"* are identical. So, the code generator produces a new method called *"setName"* with an empty body and keeps the method *"naming"* with the manually written source code. One way to avoid this problem is to rename the method *"naming"* in the generated source code to *"setName"* before editing the model. Then, during the process of the source code regeneration, the code generator maps the method name and does not generate an extra method called *"setName"* with an empty body. However, in case we edit the model by adding an extra argument in method *"setName",* original functions versions are maintained but the regeneration process introduces a duplicate method skeleton with an updated prototype. 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 full cycle development

The second approach tries to resolve the maintenance issues supporting full cycle development. In particular, the model-driven process begins, as previously, with the

model construction of the Person class. Then, the correspondent source code is generated by an appropriate code generator. Afterwards, the body of the method called “naming” is completed. Then, in case we would like to edit or extend the model, the source code is transformed by a Model Reconstructor in order to update the model according to the source code. In other words, the source code is parsed in order to identify its constructs and generate the correspondent model (i.e. Model Driven Reverse engineering). Of course, there are parts of the source code that cannot be identified (e.g. the source code of the body of a method). These parts are kept in the model as metadata. When developers finish with the model changes and the source code is regenerated, the previously manually written source code has been maintained since it has passed from the previous source code to the new source code via the model. This approach perfectly solves the maintenance issue for general purpose MDE tools as applied for instance to Papyrus and Modelio. It cannot however be deployed in case of specific mission tools. For example, in case of generative MDE tools for user-interface code generation, like *GrafiXML* [23] or *GuiBuilder* [24], it is practically impossible to recognize the widget elements by parsing manually written source code [25].

5.1.2 How our approach solves maintenance

All the attempts to solve the maintenance in general follow the logic of generating the source code and extending it in order to complete the development process as it is depicted at the top of Figure 57. Then, in case the model needs to be edited or extended during development, these approaches seek ways to shun this problem. Although, there is improvement in this way, it does not seem to be sufficient to solve the maintenance issue completely and efficiently.

Thus we started thinking of an alternative path, in which the MDE tool output would somehow remain invariant, that is in a not-editable form and the source code of the application could still grow and evolve in an unconstrained manner around it

as it is depicted in the bottom of Figure 57. In this scenario, the code to model reconstruction path is unnecessary. We will continue with the description of the previously discussed example using our approach.

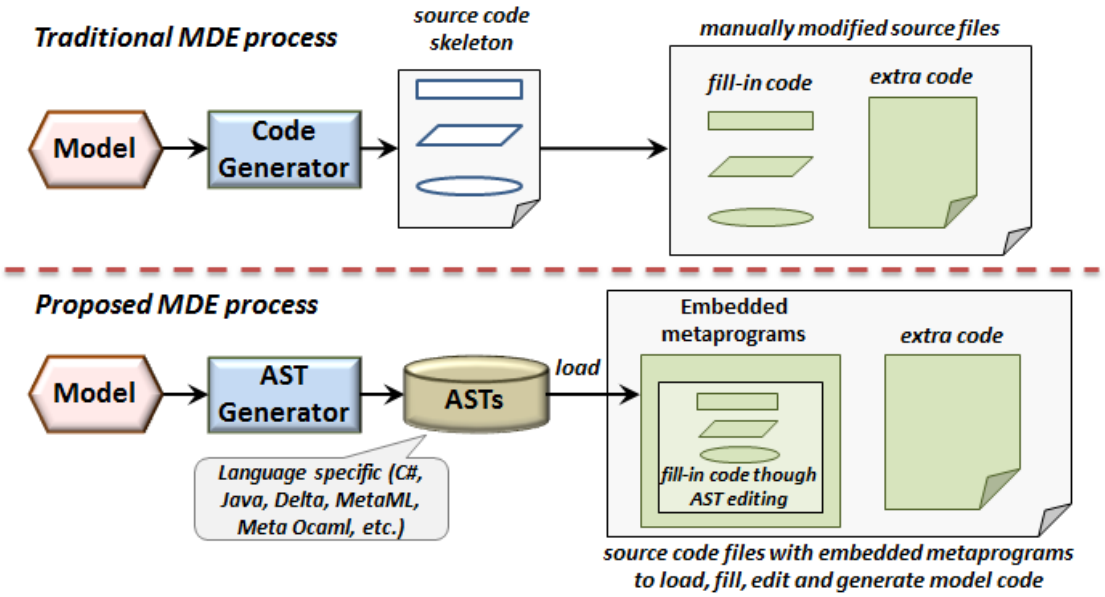


Figure 57. Top: Traditional MDE process where the generated source code files are manually updated with fill-in and extra code. Bottom: The proposed MDE process where the tool output is in AST form and the programmer deploys embedded metaprograms to load, fill, edit source code in the form of ASTs and generate a transient code version that will be integrated along with the custom application.

Using a modeling tool we design the Person class which includes the attribute *"name"* and the method *"naming"*. Then, we use an appropriate AST generator in order to generate the correspondent AST. We develop the staged code in order to load the AST of Person model code (see label 1 of Figure 58), fill the body of "naming" method (see label 2 of the Figure 58) and generate the model source code around the rest source code of application during translation (see label 1 of the Figure 58). Afterwards, the compilation process begins and the evaluation result of the staged code is depicted in label 4 of Figure 58.

During the development process, in case we decide to extend the model Person and add for example the attribute *"height",* the only thing we need to do is to use the AST generator in order to update the AST of Person. Then, we have to repeat the compilation process in order to regenerate the model source code around the rest application source code.

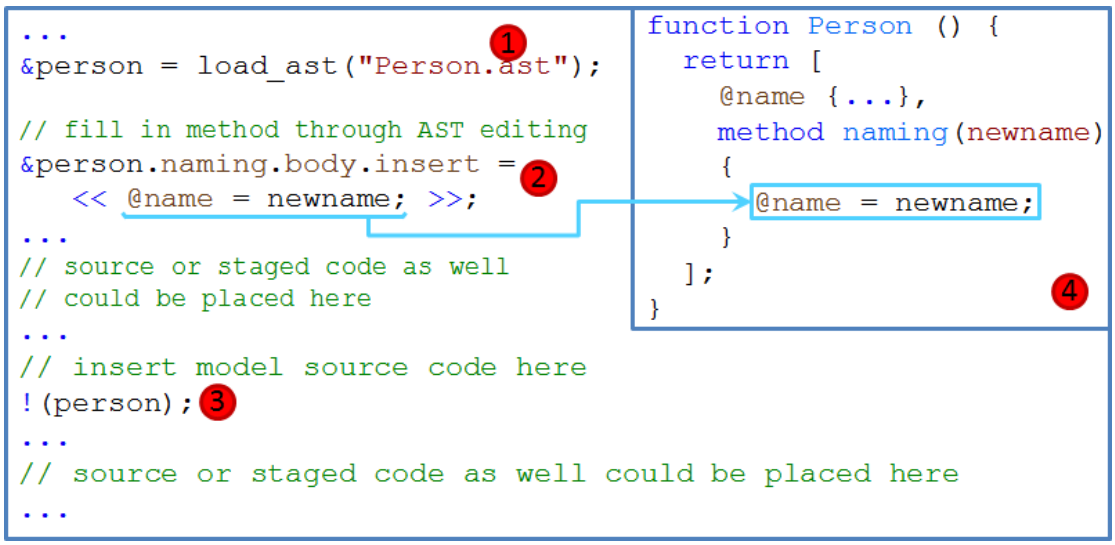


Figure 58. Developing a Person example in our approach and the result of the generated code in label 4.

In case we decide to edit the Person model and rename the method called "naming" to "setName", the AST generator will be used in order to update the AST of Person and repeat the compilation which will not succeed this time. As it is depicted in label 2 of Figure 58, `&person.naming.body`, the user data of the updated AST does not include the index naming. In particular, the index *"naming"* has been replaced with *"setName"*.

So, the staged code `&person.naming.body` has to be replaced with `&person.setName.body`. On the one hand, the compilation process will not succeed; on the other hand, the source code will not be destroyed as in the first approach described previously. The advantage in this case is that developers view the appropriate messages from the compiler (errors messages during compilation) concerning what goes wrong in the developed staged code. 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. In the following table we outline the methods which address maintenance and the case of working efficiently or not:

Approaches / Cases	Annotations	Full cycle development	Staged Code Generators
Adding new constructs in a model	Yes	Yes	Yes
Editing constructs of a model (e.g. renaming a method)	No (before the model editing, the code needs editing)	Yes	Yes
Adding new elements in constructs of a model (e.g. adding an argument in a function)	No	Yes	Yes
User-Interface Code Generation	N/A	No	Yes
Using multiple models in single development	N/A	N/A	Yes

Table 1. Comparing the approaches which deal with maintenance issues

5.2 Tradeoffs of our approach

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.

5.3 Applicability of our approach

Not all popular languages support staging, even though there are a few third party extensions such as Metaphor [48] and Mint [49]. 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 this thesis. This option is detailed under Figure 59, 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.

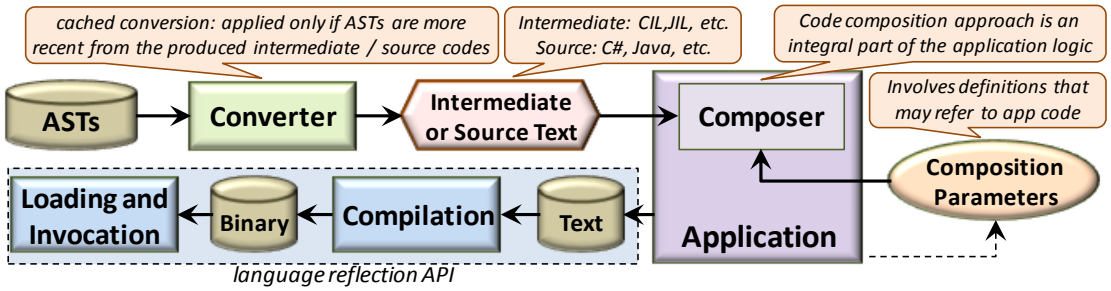


Figure 59. Applying the generative MDE process with runtime staging; the application composes intermediate or source text and then deploys the language reflection API for compilation and invocation (JIL stands for Java Intermediate Language, CIL for the Common Intermediate Language of .NET). The entire runtime conversion, composition and compilation process is cached – it is only repeated when the ASTs change, i.e. upon regeneration.

The oval of Figure 59 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.

Chapter 6

Conclusions and Future Work

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 overall 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.

In this thesis 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 order to cope with such maintenance issues. In this approach, the generator components of MDE tools need output Abstract Syntax Trees (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 also carried out several case studies to experiment and validate the engineering proposition using a compile-time metaprogramming language, an

interface builder, a general purpose modeling tool and automatic user interface. 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.

Nonetheless, it is the first time in Model-Driven Engineering that the use of metaprogamming is proposed. In this sense, an intriguing future task would be to further evaluate the proposed approach of MDE. This evaluation needs to be carried out by multiple users utilizing our approach in the development of applications and giving us much needed feedback. Hence, we will examine the effectiveness of the approach in factual circumstances. Moreover, our future plans include an extended case study in a large real-world application involving various modeling tools and legacy systems so as to better demonstrate the potential of our approach and assess its practicability.

Additionally, working for the case study of auto-generation of User Interfaces with annotated APIs, we came to realize that there are a lot of extensions which could be added in this approach. Firstly, we intend to include further expressions for the specifications. The model will evolve to a more expressive form so as to cover more mundane demands of the User-Interface cases that can occur in the development of an application. In this direction, we will further add layout specifications. Layout specifications will be far more effective than just the use of the *manipulating interface code as ASTs* operators we identified during the case study of User Interface Builder. Afterwards, we will move on to develop an appropriate software visualization tool in order to construct the annotated APIs automatically through this. Thus models will be more easily constructed and an abstract representation of this type of models will be provided.

In conclusion, it will be very interesting to research how a hybrid approach could be viable, using WYSIWYG tools and auto-generation with annotated APIs. The former has the distinct advantage that you can get explicitly what you have designed for the User Interface of the application while the latter one constructs User-

Interfaces with comparatively more speed but lags in the accuracy of the design. So, it would be interesting to research whether we could successfully combine the aforementioned advantages of both in the MDE of User Interfaces.

Bibliography

[1] Schmidt, D. C. 2006. Model-driven engineering. Computer-IEEE Computer Society, 39(2), 25-31. DOI=http://dx.doi.org/10.1109/MC.2006.58.

[2] Object Management Group. 2010. OMG Model Driven Architecture - The Architecture of Choice for a Changing World. http://www.omg.org/mda/ Accessed 7/2013.

[3] Tratt, L (2005). Compile-time meta-programming in a dynamically typed OO language. In proceedings of DLS 2005 ACM Symposium on Dynamic Languages, ACM, 49-63

[4] Sheard, T., Benaissa, Z., Martel, M. 2000. Introduction to multi-stage programming using MetaML. Technical report, Pacific Software Research Center, Oregon Graduate Institute.

[5] Taha, W. 2004. A gentle introduction to multi-stage programming. In Domain-Specific Program Generation, C. Lengauer, D. Batory, C. Consel, and M. Odersky, Eds. LNCS, vol. 3016, 30-50. Springer, Heidelberg.

[6] Delta Programming Language. 2012. Official site. http://www.ics.forth.gr/hci/files/plang/Delta/Delta.html, https://139.91.186.186/svn/sparrow (user: 'guest' with empty password). Accessed online 7/2013.

[7] Lilis, Y., Savidis, A. (2012). Supporting Compile-Time Debugging and Precise Error Reporting in Meta-Programs. In proceedings of TOOLS 2012 International Conference on Technology of Object - Oriented Languages and Systems (29 – 31 May), Prague, Czech Republic, Springer LNCS 7304, 155-170

[8] Bawden, A. 1999. Quasiquotation in Lisp. In Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation (San Antonio, 1999), 88–99. University of Aarhus, Dept. of Computer Science. Invited talk.

[9] Dybvig, R. K. 2009. The Scheme Programming Language (fourth edition). The MIT Press (ISBN 978-0-262-51298-5 / LOC QA76.73.S34D93).

[10] Ganz, S., Sabry, A., Taha, W. (2001). Macros as multi-stage computations: Type-safe, generative, binding macros in Macro ML. In Proceedings of ICFP 2001 International Conference on Functional Programming, ACM, 74 – 85

[11] MetaOCaml (2003). A compiled, type-safe multi-stage programming language. http://www.cs.rice.edu/~taha/MetaOCaml/
Accessed online 7/2013.

[12] Fleutot, F. (2007). Metalua Manual.
http://metalua.luaforge.net/metalua-manual.html.
Accessed online 7/2013.

[13] Tratt, L. 2008. Domain specific language implementation via compile-time metaprogramming, ACM Transactions on Programming Languages and Systems TOPLAS, 30(6):1-40, October 2008.

[14] The Eclipse Foundation. Eclipse Modeling Framework (EMF) (2008). http://www.eclipse.org/modeling/emf/
Accessed online 7/2013.

[15] The Eclipse Integrated Development Environment. Official site.
http://www.eclipse.org/
Accessed online 7/2013

[16] Obeo (2006) Acceleo: MDA generator
http://www.acceleo.org/pages/home/en
Accessed 7/2013.

[17] Object Management Group. (2012). Object Management Group Object Constraint Language (OCL). http://www.omg.org/spec/OCL/ISO/19507/PDF/
Accessed online 7/2013.

[18] Actifsource GmbH (2010). Actifsource Code Generator for Eclipse. http://www.actifsource.com/_downloads/actifsource_code_generator_for_Eclipse_en.pdf
Accessed online 7/2013.

[19] Badreddin, O., Lethbridge, T.C. (2013). Model Oriented Programming: Bridging the Code-Model Divide. Modeling in Software Engineering, in conjunction with ICSE 2013.

[20] 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, New York, NY, USA, 927-928. DOI = http://doi.acm.org/10.1145/1297846.1297949
Accessed online 7/2013

[21] Chalabine, M., 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 Computer Society, Washington, DC, USA, 137-146. DOI= http://dx.doi.org/10.1109/ICSE.2007.7
Accessed online 7/2013

[22] Lanusse, A., Tanguy, Y., Espinoza, H., Mraidha, C., Gerard, S., Tessier, P., Schnekenburger, R., Dubois, H., 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). University of Twente. Enschede, The Netherlands. June 23-26, 2009.

[23] Michotte, B., 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 (March 16-21), IEEE, 15-22.

[24] Sauer, S. 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: coping with diversity (UAHCI'07), Constantine Stephanidis (Ed.). Springer-Verlag, Berlin, Heidelberg, 537-546.

[25] Staiger, S. (2007). Static Analysis of Programs with Graphical User Interface. In Proceedings of 11th European Conference on Software Maintenance and Reengineering, 2007. CSMR '07. pp.252-264, 21-23 March 2007. doi: 10.1109/CSMR.2007.44

[26] Desfray, P. (2009). Modelio: Globalizing MDA. In Proceedings of the Tools and Consultancy Track of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009). University of Twente. Enschede, The Netherlands. June 23-26, 2009.

[27] Altova. 2013. UModel – UML tool for software modeling and application development. http://www.altova.com/umodel.html
Accessed online 7/2013.

[28] Systems Modeling Language SysML v1.3 2012 by OMG. Official site http://www.omgsysml.org/
Accessed online 7/2013.

[29] Business Process Model and Notation BPMN by OMG. Official site http://www.bpmn.org/
Accessed online 7/2013.

[30] Sparx Systems. 2000. Enterprise Architect – Visual Modeling Platform. http://www.sparxsystems.com/products/ea/index.html
Accessed online 7/2013.

[31] Gentleware AG. 2007 Apollo for Eclipse – UML Modeling tool Extension to Eclipse. http://www.gentleware.com/apollo.html
Accessed online 7/2013.

[32] Graphical Modeling Framework GMF, an Eclipse modeling project
http://wiki.eclipse.org/GMF
Accessed online 7/2013.

[33] wxFormBuilder. 2006. A RAD tool for wx GUIs.
http://sourceforge.net/projects/wxformbuilder/
Accessed online 7/2013.

[34] Wx Widgets. A widget toolkit and tools library for creating GUIs for cross-platform applications.
http://www.wxwidgets.org/
Accessed online 7/2013

[35] USIXML. USer Interface eXtensible Markup Language
http://www.usixml.org/en/what-is-usixml.html?IDC=236
Accessed online 7/2013

[36] Glade – A User Interface Designer
https://glade.gnome.org/
Accessed online 7/2013

[37] GtkBuilder – Build an interface from an XML UI definition
https://developer.gnome.org/gtk3/stable/GtkBuilder.html
Accessed online 7/2013

[38] wxGlade – A GUI builder for wxWidgets
http://wxglade.sourceforge.net/
Accessed online 7/2013

[39] wxPython - A blending of the wxWidgets C++ class library with the Python programming language. http://www.wxpython.org/
Accessed online 7/2013

[40] wxDesigner – Dialog editor and RAD tool for wxWidgets
http://www.wxdesigner-software.de/ Accessed online 7/2013

[41] Blend - A user interface design tool.
http://www.microsoft.com/expression/eng/
Accessed online 7/2013

[42] XAML - A declarative markup language.
http://msdn.microsoft.com/en-us/library/ms752059.aspx
Accessed online 7/2013

[43] K. Doddapaneni, E. Ever, O. Gemikonakli, I. Malavolta. (2012). A Model-Driven Engineering Framework for Architecting and Analysing Wireless Sensor Networks. SESENA 2012: Zurich, Switzerland Third International Workshop on, vol., no., pp.1,7, 2-2 June 2012 doi: 10.1109/ SESENA.2012.6225729.

[44] A.W.O. Rodrigues, F. Guyomarc'h, J.-L. Dekeyser. An MDE Approach for Automatic Code Generation from MARTE to OpenCL. (2011). http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6171148
Accessed online 7/2013

[45] Development of Intelligent User Interfaces and Games.
Lecture of Computer Science Department, University Of Crete. Official site
http://www.csd.uoc.gr/~hy454/slides.html
Accessed online 7/2013.

[46] User Interface Description Language UIDL. Official Website
http://www.uidl.net/
Accessed online 7/2013.

[47] XML Based Resource System (XRC). Official Website
http://docs.wxwidgets.org/trunk/overview_xrc.html
Accessed online 7/2013.

[48] Palmer, Z., Smith, S. F. (2011). Backstage Java: Making a difference in metaprogramming. In proceedings of OOPSLA 2011 International Conference on Object Oriented Programming Systems, Languages and Applications, ACM, 939-958.

[49] Westbrook, E., Ricken, M., Inoue, J., Yao, Y., Abdelatif, T., 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, NY, USA, 400-411.DOI=http://doi.acm.org/10.1145/1806596.1806642.