Improved Model-Driven Engineering of User-Interfaces with Generative Macros

Anthony Savidis[1,2], Yannis Valsamakis[1], and Yannis Lilis[1]

[1] Institute of Computer Science, FORTH
[2] Department of Computer Science, University of Crete, Greece
{as,jvalsam,lilis}@ics.forth.gr

Abstract. Model-driven engineering entails various modeling, abstraction and specialization levels for user-interface development. We focus on model-driven tools generating user-interface code, either entire or partial, providing a tangible basis for programmers to introduce custom refinements and extensions. The latter introduces two maintenance issues: (i) once the generated code is modified the source-to-model extraction path, if supported, is broken; and (ii) if the model is updated, code regeneration overwrites custom changes. To address these issues we proposed an alternative path: (i) instead of directly generating code, the model driven tool generates source fragments in the form of abstract syntax trees (ASTs) as XML files; (ii) the application deploys compile-time metaprogramming to manipulate, generate and insert code on-demand from such ASTs, using calls similar to macro invocations. The latter leads to improved separation of concerns: (a) the application programmer controls when and where interface source is generated and integrated in the application source; and (b) interface regeneration overwrites no source code as it only produces ASTs that are manipulated (input) via generator macros.

Keywords: Model-Driven Development, Model-Based User-Interfaces, Code Generation, Compile-Time Metaprogramming.

1 Introduction

In general, model-driven engineering (MDE) of user interfaces [3] involves tools, models, processes, methods and algorithms addressing the demanding problem of automated user-interface engineering. An important authoring requirement for MDE tools is to involve notions and concerns inherent in the design domain, typically including tasks, user profiles, context characteristics, interaction controls, abstract behaviors, and input events. Then, a target implementation is incrementally derived, usually with an intermediate transition from the modeling domain to an instantiation domain that is in most cases platform independent. This discipline is outlined under Fig. 1, showing the typical specialization from abstract to platform that most MDE tools adopt, and the shift from abstract to concrete implying a sort of transformation.

C. Stephanidis and M. Antona (Eds.): UAHCI/HCII 2014, Part I, LNCS 8513, pp. 137–148, 2014.
© Springer International Publishing Switzerland 2014

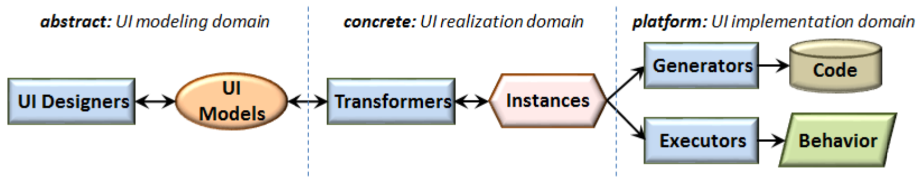

Fig. 1. High-level overview of the model-driven user-interface development process showing tool roles with respective inputs and outputs

Currently, many MDE tools emphasize the delivery of user-interface source code (like wxFormBuilder and Microsoft Visual Studio), thus encompassing user-interface code generators, sometimes capable to cater for varying programming languages and target platforms. Alternatively, MDE tools may provide executors, usually falling in the domain of interaction interpreters, which directly offer the required end-user dialogue (such as the XCode Interface Builder). The latter actually interpret the various produced concrete instances and their accompanying internal interface representations, as resulting from the transformations on the abstract modeling domain. In our work we generally focus on MDE tools that eventually generate user-interface source code, such tools ranging from common interface builders to entire model-driven application framework generators.

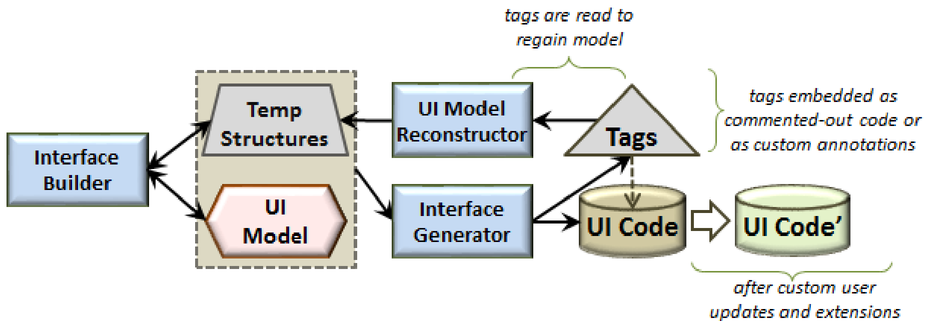

Fig. 2. General architecture of interface builders involving: (1) interactive editing with the builder; (2) code generation from an explicit interface model or temporary structures; and (3) only when temporary structures are used, tags are inserted in the source code.

In the context of our case study we considered interface builders – their general process diagram is outlined under Fig. 2. Interface builders usually support an explicit interface model, usually in a custom user-interface description language (UIDL). Alternatively, they may rely on special tags carrying model information, such tags explicitly embedded in the generated source as commented-out code. Through tags, no interface model is explicitly stored, except temporary structures which are available only during authoring time.

We continue with the identification of the maintenance problem inherent in user-interface code generation, and then brief the key contributions of our work to address this issue.

1.1 Identification of the Problem

MDE tools, whether finally offering user-interface source code generation or an execution system at the platform level, can hardly address all required aspects of an interactive user-interface. As a result, custom user-interface source code amendments and modifications are always anticipated. Even there are evolved tools in Model-Driven of user-interfaces area; none of them can completely construct the user-interface of applications. In our context we focus only on MDE tools generating user-interface source code because we consider they are more flexible by supporting evolution and customization of the user-interface directly at the source level.

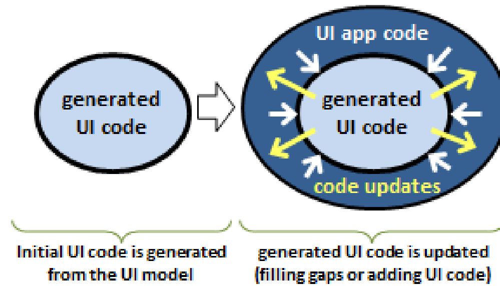

Fig. 3. Typical growth of user-interface application code around generated user-interface code with custom extensions and updates, eventually leading to bidirectional dependencies.

With such MDE tools, the typical lifecycle of the generated user-interface code is outlined under Fig. 3. In particular, typical updates relate to user-interface application functionality importing and invocation, event handling extensions, custom user-interface management logic, and linkage to third-party libraries that are not known to the MDE tool. This situation very quickly results into many bidirectional dependencies as indicated at the right part of **Fig. 3**.

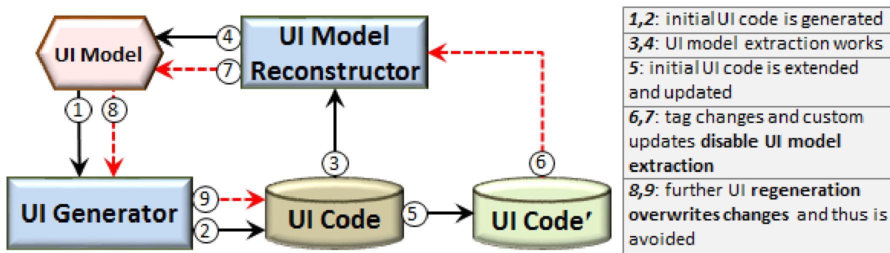

Fig. 4. The primary maintenance issue inherent in user-interface source generation under model-driven development

The problem with this scenario is that it introduces serious maintenance issues (see Fig. 4). Initially, once the user-interface code is not changed, user-interface regeneration and user-interface model reconstruction is well-defined (steps 1 to 4). In other words, the MDE tool works perfectly for both steps of the processing loop. However,

once the generated UI code is updated (step 5), two problems directly appear. Firstly, tag editing and misplacing may break model reconstruction (steps 6 and 7), while the manually inserted UI logic outside the MDE tool causes a model-implementation conflict. Secondly, source regeneration overwrites all manually introduced updates (steps 8 and 9). For real life applications of a considerable scale the latter may lead to adoption of the MDE tool only for the first version, or worse avoiding using an MDE tool at all.

1.2 Primary Contributions

Our main contribution is an inversed responsibility model for generator MDE tools where: (a) UI generation takes place only in the form of ASTs; and (b) the actual code generation is applied on-demand and in-place through metaprograms (macros) that are included in the implementation of the interactive system and are evaluated at compile-time (i.e. during the build process). This approach, not only resolves the maintenance issues of traditional UI generators, but also sets user-interface code manipulation as a first-class concept in user-interface management and reveals the value of metaprogramming languages in the engineering of interactive systems.
Overall, we propose an improved process where the MDE tool outcome is read-only, decoupled from UI code generation, letting interactive applications directly deploy and manipulate code fragments, instead of being built around them. In this context, we discuss the most common composition practices on user-interface code fragments through ASTs as consolidated from our case study.

2 Related Work

Several UI source code generators exist in the arena of MDE tools and most of them are incorporated into UI Builders. Some of them are GrafiXML [10], GuiBuilder [11], GtkBuilder [12], wxFormBuilder [9] etc none of which has addressed the maintenance issues we have discussed.

Furthermore, there are relevant works that partially address the maintenance issues not for UI code generation but for general purpose source code generation. In particular, there are two different ways in which the problem has been approached.

The first approach includes special tags or annotations which are inserted within the generated source code. Developers may further edit such annotations to specify whether certain parts of the source code should be maintained or not upon regeneration. However, free editing may cause tag misplacement and thus result in manual updates being discarded upon regeneration. In the one hand, tags address the problem; on the other hand there is extra responsibility for developers. Tools which adopt this approach are EMF [13], Acceleo [14], Actifsource [15] etc.

The second approach is based on the full MDE development cycle allowing both model-to-source and source-to-model transformations. For the latter, they parse source files locating specific code structures (e.g. Classes, Attributes, Operations etc.) in order to regenerate the model, while treating any additional code they include as

metadata. This is an important step towards resolving the maintenance issues; however, it cannot be applied in case of MDE tools for UI code generation, because it is practically impossible to recognize the widget elements by parsing manually written source code. Tools which adopt this approach are Papyrus [16] and Modelio [17].

3 Staged Metaprograms

Generally, metaprogramming relates to functions which generate code, i.e. programs producing other programs, while metaprogramming languages take the task of code generation and support it as a first-class language feature. This is a sort of reification of the language code generator enabling programmers write code which generates extra source code. When available as a macro system before compilation, the method is known as compile-time metaprogramming. Alternatively, if offered during runtime, usually built on top of the language reflection mechanism, it is called runtime metaprogramming. We focus on compile-time metaprogramming being more powerful to 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 [2, 4]. In our work we use Delta [1], a recent publicly available dynamic object-oriented language, its wx widgets library, and its compile-time metaprogramming extension [5, 6]. Popular meta-languages include Lisp, Scheme, Macro ML [7], Meta OCaml [8], Meta Lua and Converge.

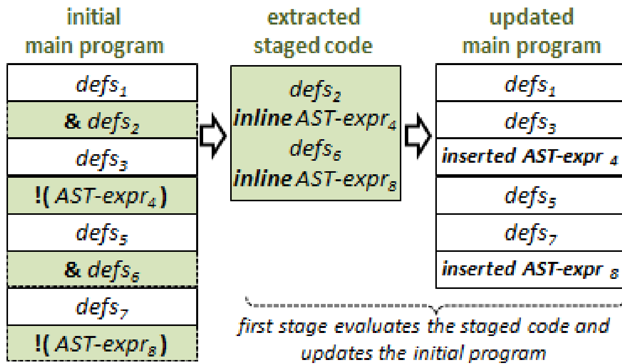

Fig. 5. Metaprogram evaluation as a compilation stage

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.

As shown under Fig. 5, in the first stage the compiler: (i) collects all scattered meta-code into a single metaprogram; (ii) evaluates the program while internally

recording the output 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.

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

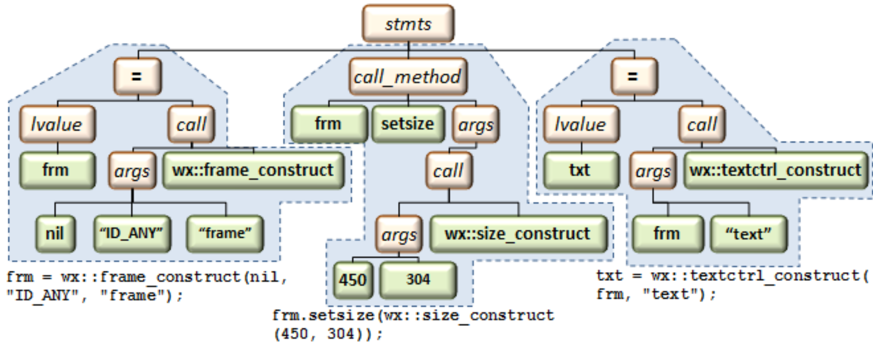

Fig. 6. Example of an abstract syntax tree for three statements using the wx widgets library: (i) *left*: creating a frame widget; (ii) *middle*: setting its size; and (iii) *right*: creating a text widget

The first line is normal code, a typical directive to import the wx widgets 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 Fig. 6. The third line inserts the code implied by this AST into the main program. As a result, after the first stage, and before normal compilation, the main program is:

```
using wx;
frm = wx::frame_construct(nil, "ID_ANY", "calculator");
frm.setsize(wx::size_construct(450, 300);
txt = wx::textctrl_construct(frm, "text");
```

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.

4 Improved Model-Driven Process

The primary motivation for our work has been the serious user-interface source code maintenance issue inherent in model-driven UI code generators. Although we needed to avoid this problem, in the mean time we wished to retain the powerful generational character of MDE tools. Thus we started thinking of an alternative path, in which: (i)

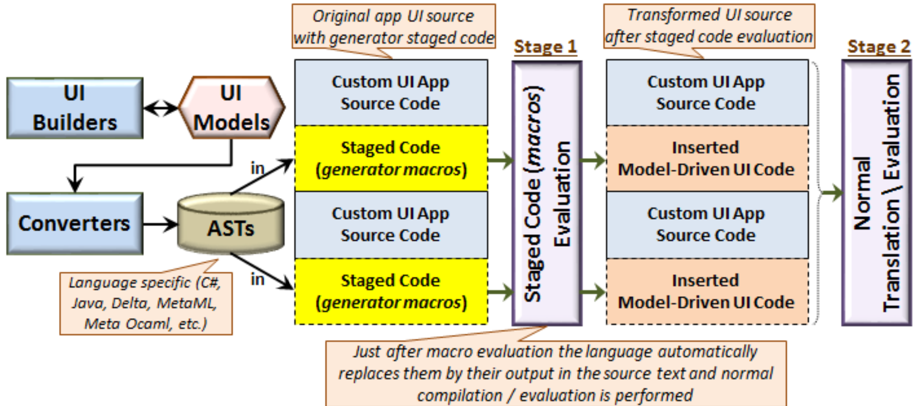

Fig. 7. The improved model-driven process with inverted responsibility: user-interface programmers deploy generator macros to produce resulting code on-demand and in-place without affecting the originally produced ASTs by the MDE tools

the MDE tool output would somehow remain invariant; and (ii) the source code of the interactive application could still grow in an unconstrained manner around it. This led us to the idea of bringing metaprogramming into the pipeline by enabling interface engineers algorithmically manipulate the generated interface code including: loading, processing and transforming using macros that are evaluated during build time. Such a process is detailed under Fig. 7.

As shown, we suggest that the MDE tools should generate the concrete produced user-interfaces in a user-interface description language (UIDL), such definitions naturally involving all the necessary structural, algorithmic and event management details, but in a language neutral form. This is only proposed to allow language independent MDE tools, but is not strict, meaning MDE tool developers may choose to directly produce ASTs for a specific target programming language. The general approach for code manipulation and insertion using ASTs is the one earlier described and relates to compile-time metaprogramming languages, involving two stages that are also depicted under Fig. 7: meta-code evaluation (stage 1), and normal compilation (stage 2).

5 Development Case Study

To test our approach and assess its expressive power and engineering validity, we have carried out a case study. We have adopted wx Form Builder [9], 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 constructed a full-scale scientific calculator application. 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 had to build an appropriate converter, following the proposed approach at the left of Fig. 7. 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 wx Form Builder. In-between this process we repeated many times reloading of the visual models and regenerating of the XRC files, 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.

5.1 Manipulating Interface Code as Abstract Syntax Trees

The goal of our case study was 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. In Fig. 8, two of the composition scenarios which have been implemented are outlined.

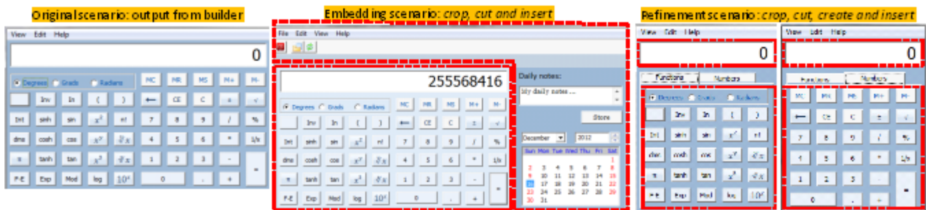

Fig. 8. Two example scenarios (middle, right) of user-interface source code composition relying on AST manipulation on top of the original GUI authored with the interface builder (left); updates on the two scenarios are automated and are *directly* remapped on top of the original GUI by simply performing recompilation

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 where event management code is injected within a user-interface construction code snippet.

In the following table, we enumerate and briefly describe 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 previous code fragment.

Table 1. Manipulation Operators for User-Interface

Operator	**Description**
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. It was needed to extract the code creating the numeric and function pads of the calculator case study. When directly followed by an insert operation it implements re-parenting.
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.
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.
Create	This is not an operator on the input source code fragments per se. 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 study the latter concerned the tab-box with the Functions and Numbers entry (see Figure 8, right part).
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.
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 UI 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.
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.

5.2 Composing Interface Code In-Place and On-Demand

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.

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 Fig. 8:

```
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
```

The colon is used to indicate 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, "calcFrame");                 ← drop the outer Frame inserted by wx Form Builder      (1)
Tree::Insert(                                  ← insert an application event handler for = button
  calc, "buttEqual", "EVT_COMMAND_BUTTON_CLICKED",
  "CalcApp::OnEqual"                           ← handler function provided by the application
);                                                                                                     (2)
Tree::Insert(                                  ← insert an application event handler for + button
  calc, "buttAdd", "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, "panelNums");  ← cut the code constructing the numeric panel          (3)
local funcs   = Tree::Cut(calc, "panelFuncs"); ← cut the code constructing the functions panel
local panel   = Tree::Get(calc, "panelMain");  ← 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   (4)
local numsTab  = << code here to create the numbers tab entry   >>;
local funcsTab = << code here to create the functions tab entry >>;
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
```

Fig. 9. 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 Fig. 9, 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 Fig. 9 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 **Fig. 9**) and compose them via `Tree::Insert` with the ASTs earlier extracted from the calculator code.

6 Summary and Conclusions

Currently, MDE of User-Interfaces represents a domain of very powerful development tools for rapid development of interactive systems. Their evolution in the last decade consolidated the disciplined view of model-based user-interface generation as a transformation process from abstract to concrete models, eventually down to the physical platform level. Generational MDE tools support the production of concrete user-interface implementations directly at the source code level. Such a facility is overall very helpful, powerful and flexible for user-interface programmers. However, it also causes maintenance issues once extensions and updates are manually introduced over the generated user-interface.

To cope with such maintenance issues we propose the exploitation of the metaprogramming language facilities and suggest an improved model-driven code of practice relying on the manipulation of user-interface code fragments by clients directly as data. 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 a case study to experiment and validate the engineering proposition using a publicly available compile-time metaprogramming language and an interface builder. Overall we were truly impressed by the compositional flexibility which allowed us to safely and easily manipulate and extend the produced interface without suffering from maintenance issues. We believe our work reveals the chances by combining metaprogramming and generational MDE user-interface engineering tools, and anticipate more efforts to further exploit this field.

References

1. Savidis, A.: Delta Programming Language (2012), `http://www.ics.forth.gr/hci/files/plang/Delta/Delta.html` (accessed february 2014)

2. Taha, W.: A gentle introduction to multi-stage programming. In: Lengauer, C., Batory, D., Blum, A., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 30–50. Springer, Heidelberg (2004)
3. Schramm, A., Preußner, A., Heinrich, M., Vogel, L.: Rapid UI Development for Enterprise Applications: Combining Manual and Model-Driven Techniques. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 271–285. Springer, Heidelberg (2010)
4. Sheard, T., Benaissa, Z., Martel, M.: Introduction to multi-stage programming using MetaML. Technical report, Pacific Software Research Center, Oregon Graduate Institute (2000)
5. Lilis, Y., Savidis, A.: Implementing Reusable Exception Handling Patterns with Compile-Time Metaprogramming. In: Avgeriou, P. (ed.) SERENE 2012. LNCS, vol. 7527, pp. 1–15. Springer, Heidelberg (2012)
6. Lilis, Y., Savidis, A.: Implementing Reusable Exception Handling Patterns with Compile-Time Metaprogramming. In: Avgeriou, P. (ed.) SERENE 2012. LNCS, vol. 7527, pp. 1–15. Springer, Heidelberg (2012)
7. Foley, J., Kim, W.C., Kovacevic, S., Murray, K.: Defining Interfaces at a High Level of Abstraction. IEEE Software 6(1), 25–32 (1989)
8. MetaOCaml, A compiled, type-safe multi-stage programming language (2003), http://www.metaocaml.org/
9. wx Form Builder (2006), A RAD tool for wx GUIs, http://sourceforge.net/projects/wxformbuilder/ (accessed online January 2014)
10. Michotte, B., Vanderdonckt, J.: 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, pp. 15–22. IEEE (2008)
11. Sauer, S., Engels, G.: Easy model-driven development of multimedia user interfaces with guiBuilder. In: Stephanidis, C. (ed.) HCI 2007. LNCS, vol. 4554, pp. 537–546. Springer, Heidelberg (2007)
12. GtkBuilder – Build an interface from an XML UI definition, https://developer.gnome.org/gtk3/stable/GtkBuilder.html (accessed online January 2014)
13. The Eclipse Foundation. Eclipse Modeling Framework, EMF (2008), http://www.eclipse.org/modeling/emf/ (accessed online January 2014)
14. Obeo (2006), Acceleo: MDA generator, http://www.acceleo.org/pages/home/en (accessed online January 2014)
15. Actifsource GmbH (2010), Actifsource Code Generator for Eclipse, http://www.actifsource.com/_downloads/actifsource_code_generator_for_Eclipse_en.pdf (accessed online January 2014)
16. Lanusse, A., Tanguy, Y., Espinoza, H., Mraidha, C., Gerard, S., Tessier, P., Schnekenburger, R., Dubois, H., Terrier, F.: 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)
17. Desfray, P.: 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)