Customizing the Visualization and Interaction for Embedded Domain-Specific Languages in a Structured Editor

Dimitar Asenov
Department of Computer Science
ETH Zurich
dimitar.asenov@inf.ethz.ch

Peter Müller
Department of Computer Science
ETH Zurich
peter.mueller@inf.ethz.ch

***Abstract*—Large software projects are often based on libraries that provide abstractions for a particular domain such as writing database queries, staging, or constraint solving. The API provided by such a library can be considered a domain-specific language within the implementation language of the library, a so-called internal or embedded domain-specific language (eDSL). Embedding a DSL leverages the tool infrastructure of the host language, but also restricts the syntax and IDE support to that of the host language. This restriction prevents programmers from using convenient specialized notations and, thus, has a negative effect on their productivity. To address this problem, we outline concepts for a structured code editor that enable developers of eDSLs to customize how eDSL code is rendered and what interactions are available. We demonstrate the benefits of our approach by customizing a structured editor for the .NET Code Contracts API. Our prototype shows in particular that we can customize many aspects of visualization and interaction with little effort.**

***Keywords*—programming environments, embedded domain-specific languages, structured editors, editor customization, visual programming, human-computer interaction**

I. Introduction

Large software systems frequently make use of domain-specific languages (DSLs) to describe specific aspects of the system in a concise way. Examples include DSLs for specifying database queries, business processes, or security policies. Some DSLs, so-called *external* DSLs such as SQL in the domain of database queries, define their own (textual or graphical) notations, which allow programmers to concisely express high-level concepts from a particular domain. Code written in such DSLs has been shown to be easier to comprehend [1] than equivalent code in a general-purpose language. However, external DSLs require their own tool support such as parsers, code generators, etc., which makes their development costly.

To avoid this drawback, domain-specific abstractions are often implemented as libraries in a general-purpose language. The API provided by such a library can be considered a DSL within a general-purpose host language, a so-called *internal* or *embedded* domain-specific language (eDSL), such as Querydsl[1] for database queries in Java. Embedding a DSL leverages the tool infrastructure of the host language, but also restricts the syntax and IDE support to that of the host language. Even though some host languages such as Python and Scala provide ways to customize syntax, for instance, through operator overloading, they do not fully support specialized notations. For instance, SQL's `WHERE` clause is implemented as a method call in Querydsl and treated as such by the IDE. Therefore, the IDE offers neither specific visualizations nor interactions for the embedded DSL code, which has a negative effect on programmer productivity.

In this paper, we propose concepts that allow structured code editors to be customized in order to visualize and manipulate any language construct depending on its program context and purpose. In particular, the developer of an eDSL may customize how eDSL code is visualized and manipulated within client code, for instance, to display a call to Querydsl's `where` method in the familiar SQL syntax. This gives embedded DSLs much of the flexibility of external DSLs, while retaining the benefits of operating inside a host language. We have implemented the proposed concepts in a structured editor and demonstrate their usefulness by customizing its visualizations and interactions for the .NET Code Contracts library.

II. Key Concepts of Customization

The following concepts enable API designers to easily customize the look and feel of the eDSLs they provide.

1. Decouple the storage format from visualization and interaction. In traditional source code editors, the program text is displayed and manipulated directly. This tight coupling between the program's storage format and the way it is rendered and edited restricts flexibility, especially for eDSLs, which are limited by the syntax of the host language. A prerequisite for customized visualizations and interactions is, thus, to decouple the storage format of a program from the representation that is used for rendering and editing. Editors should operate on an abstract syntax tree (AST), even when the visualization and interactions are textual.

2. Choose visualizations based on context. In a customizable editor, program fragments and language constructs can be visualized in different ways. The kind of construct being rendered should not be the sole determinant of the visualization to use. Developers of eDSLs should have the freedom to define suitable visualizations based on the following additional factors:
***Construct instance*:** What is the specific instance of the programming construct? For example a method call could be rendered differently based on its target method; a string

[1] www.querydsl.com

978-1-4799-0369-6/13/$31.00 ©2013 IEEE

constant that represents a URL might be rendered as a hyperlink, permitting additional interactions.
***Structural context*:** Where in the structure of the program is this construct? For example, an expression inside a call to Querydsl's `where` method might be shown differently than the same expression outside a Querydsl context.
***Visualization context*:** What visualizations will appear alongside the construct that is being rendered? For example when rendering Querydsl calls together, their corresponding parts could be visually aligned.
***Visualization purpose*:** Why is this construct being rendered, what is the purpose of the current task? For example, when debugging calls to Querydsl's `where`, it could be useful to show information such as the number of matching rows. This might not be needed if one is simply exploring unfamiliar code.
***Personal preferences*:** Has the developer requested a particular visualization? For example, they might prefer to represent regular expressions as automata instead of text.

3. Allow customization of interactions and make each visualization interactive. It is essential that eDSL designers are able to define interactions for the visualizations they create such as new shortcuts, options, context-menus, commands, etc. Providing merely "read-only" visualizations might help with program comprehension but not with manipulation; programmers would need to switch to the default representation of the host language for editing, which is cumbersome and reduces the benefits of DSLs. Also existing visualizations may benefit from new, customized interactions. For example one could create a new interaction for string literals that simplifies the input of file paths by overriding the TAB key to perform a name match, like a command terminal.

4. Make creating simple customizations easy, facilitate composition, and enable advanced customizations. The editor should permit eDSL designers to develop visualizations and interactions, and ship them together with their libraries. These customizations should go beyond simple style files for syntax highlighting and not require detailed knowledge of the editor implementation. For instance, eDSL designers should be able to quickly implement common visualizations such as text, boxes, icons, and lists, and create new visualizations by composing existing ones. The interactions of new compositions should automatically emerge as the aggregation of the interactions of their parts. This will greatly reduce the efforts required to implement customizations. Moreover, developers who are well familiar with the APIs provided by the editor should be empowered to create advanced customizations.

III. Prototype

To demonstrate the applicability of the proposed concepts we use Envision—our prototype of a structured code editor. The open source implementation and a video showing the customizations discussed in this paper are available online[2].

Envision features a Model-View-Controller architecture where the model is similar to an AST, and any subtree can be visualized in arbitrary ways. The default is to use text for low-level constructs such as expressions, and graphical notations for top-level constructs such as classes and methods. Textual and graphical visualizations are treated equally in Envision, and can be combined in different ways. The easiest one is to nest them by creating a new composite visualization that defines the layout used for arranging its subcomponents. The layout is specified declaratively, similar to GUI frameworks such as Qt and Swing. Subcomponents can be decorative elements (e.g., labels) or visualizations of AST nodes. Most visualizations in the screenshots from Sec. IV are designed using nesting, but visualizations can also be combined non-hierarchically as shown in Sec. IV-2. Registering a visualization in the system includes specifying the context in which it is applicable. All factors specified in concept 2 from Sec. II can be used to determine the context. When rendering an AST node, Envision chooses a visualization by scoring all visualizations applicable in the current context and picking the best one.

Each visualization has a controller that defines its interactions. Controllers can be reused, making it easier to create new visualizations. For example, Envision provides a text-like cursor and a parsing module that can be used by any visualization. We use them to enable the manipulation of expression AST nodes using the keyboard, like a standard text editor. When editing expressions, the AST is first unparsed into the host language, the text is edited, and the result is reparsed. Invalid text results in a special Error AST node. To enable text editing for a new eDSL visualization, its designer needs to define a bidirectional mapping between the visualization's components and text. For visualizations resembling text, only a few lines of code are needed. In general, eDSL designers can add new controllers or use the extension mechanisms of existing ones.

To take advantage of these customization mechanisms, eDSL designers may either just annotate the eDSL implementation as shown later, or they may define advanced customizations in a separate plug-in for Envision. Based on our own experience in using and improving the implementation, we find that little effort is required to implement new visualizations and interactions. For example, when composing visualizations, the system automatically provides a cursor that enables keyboard-based navigation and selection, similar to those in a text editor. Typically this desired behavior requires no extra code in new visualizations, therefore reducing their implementation effort.

IV. Case Study

To demonstrate the usefulness of some of the proposed customization features, we have customized Envision for the .NET Code Contracts library[3]. Code Contracts is an eDSL that allows programmers to annotate code with assertions such as method pre and postconditions. Most assertions are expressed via calls to static methods of a class `Contract` as illustrated by Fig. 1. Due to the eDSL approach, the code annotated with Code Contracts remains standard C# code and can be handled by the standard compiler. Additional tools support documentation generation, run-time checking, static analysis, and automatic test case generation.

The small example in Fig. 1 already demonstrates three issues caused by Code Contracts being an embedded DSL. (A) Code Contracts are quite verbose compared to contract languages with designated syntax such as Eiffel. (B) The notation is sometimes inconvenient since it needs to satisfy the rules of the host language; for instance, the call to `Result`

[2]www.pm.inf.ethz.ch/research/envision

[3]research.microsoft.com/en-us/projects/contracts

```
public int factorial(int x) {
  Contract.Requires(x >= 0);
  Contract.Ensures(Contract.Result<int>()>0);
  return x <= 1 : 1 ? x * factorial(x - 1);  }
```

Fig. 1. A C# factorial method. The first two statements are calls to Code Contract methods to express the pre and postcondition of the method. The call to `Result` in the postcondition refers to the return value of the method; its type argument is needed to satisfy the type system of the host language.

requires a type argument even though it is always the result type of the enclosing method. Even bigger inconvenience incurs for interfaces and out-parameters, as we illustrate later. (C) Calls encoding pre and postconditions occur within the method body, although conceptually they belong to the client-visible method signature. In the rest of this section, we will show how to customize Envision in three steps to address these issues and to give Code Contracts the convenience of native language support despite being an embedded DSL.

1) Custom visualizations for contract methods: Following concept 1 from Sec. II, we do not simply visualize calls to contract methods in C# syntax, but apply custom visualizations (here, based on the target method of the call). To address issue (C) above, we visualize these calls as part of the method signature, separated from the body by a dashed line. Associating contracts with the method signature is not only conceptually sound, but may also have other positive effects. For example, a semantic zoom showing only method signatures would now also include the contracts. Moreover, to address issue (A) above, we visualize calls to contract methods using a keyword style. The effect of these customizations is shown in Fig. 2.

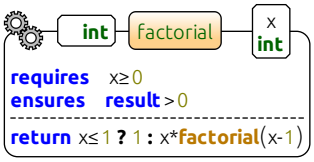


Fig. 2. The factorial method from Fig. 1 with custom visualizations. Contracts are visualized using keywords and visually separated from the method body.

Displaying contracts as part of the method signature is achieved by composing customizations according to concept 4. All visualizations in Envision have optional *add-ons*, which are simply other visual items. Add-ons allow one to display additional information within a visualization. For contracts, we created an add-on for methods that displays additional items in the signature. This add-on reuses the existing visualization of the (entire) method body. To avoid that the method body is visualized twice, we apply *list filtering* as a second customization. Envision allows visualizations of list nodes to filter which elements get displayed. For method contracts, we installed a filter for statement lists that is sensitive to the visualization context (see concept 2 from Sec. II): If the visualization appears in a method signature, only contract calls are displayed; otherwise, everything except contract calls is shown. Together, these two customizations render contracts as part of the method signature.

To apply the keyword style for contract methods, we apply a built-in visualization that shows method calls as keywords instead of using the normal method name. To change the visualization of all calls to a method, the eDSL designer simply needs to annotate the method definition with the attribute `EnvisionKeywordVisualization(style)`. Here, *style* identifies a style in an XML file that allows visualizations to be easily configured without recompilation. It is possible to change the text, font, color, background, placement, and other parameters. It is also possible to specify an icon instead of text. The complete style for preconditions is shown in Fig. 3; it specifies the keyword **requires** and uses defaults for all other parameters of the style.

```
<style prototypes="default"><keyword>
    <symbol>requires</symbol>
</keyword></style>
```

Fig. 3. The style for precondition visualizations.

To illustrate the flexibility of visualization styles, consider the example in Fig. 4. The second postcondition refers to the value of the `size` field, at the time of the method call, the so-called *old* value of `size`. In C# syntax, the old value is denoted by `Contract.OldValue(size)`. Again, we apply a context-dependent visualization, which renders calls to `OldValue` using the superscript keyword `OLD`.

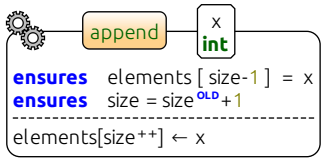


Fig. 4. A keyword visualization with a different style.

Note that all of these visualizations are fully interactive following concept 3: contracts can be added, edited, and removed directly in the signature.

2) Custom visualizations for interfaces: Since interface methods do not have bodies, there is no place in the interface definition where one could write calls to contract methods. To work around this limitation, Code Contracts force developers to create a dummy contract class that implements the interface and whose sole purpose is to contain the contracts. This special contract class and the interface are linked by attributes as shown in Fig. 5. This solution requires a lot of boilerplate code and makes reading the contracts of an interface difficult.

```
[ContractClass(typeof(ICalcContract))]
interface ICalc { int op(int x, int y);}

[ContractClassFor(typeof(ICalc))]
abstract class ICalcContract : ICalc {
   int ICalc.op(int x, int y){
      Contract.Requires( x != y );
      return 0;
   }  }
```

Fig. 5. An interface with an associated contract class. The attributes in square brackets link the interface and the class. The **return** statement is necessary to satisfy the C# compiler.

To address issue (B) above and shield the programmer from this inconvenient notation, we perform two customizations. First, we create another add-on for methods. It is active within

the context of interface method declarations and displays the contracts from the associated contract class, as shown in Fig. 6. This example illustrates that visualizations may depend on the entire AST (see concept 2 from Sec. II), in this case, code contained in a different class. Again, this visualization is fully interactive according to concept 3: programmers may edit the contracts using the add-on, as if the contracts were specified in the interface itself. Second, we hide the contract class entirely as it is no longer necessary and also omit the attribute in the interface referring to that class.

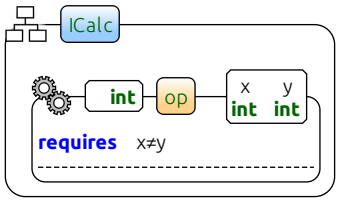
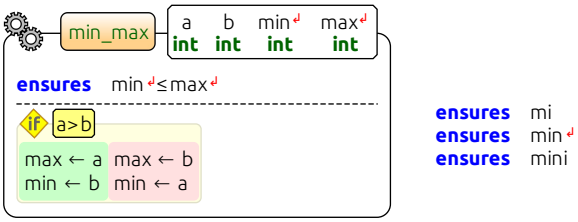


Fig. 6. The `ICalc` interface from Fig. 5 with a contract visualization add-on. The corresponding contract class is hidden as it is no longer needed.

3) Custom interactions for contract methods: Postconditions in Code Contracts may refer to out-parameters. Since postconditions appear at the beginning of the method bodies, the C# compiler complains that the value of an out-parameter appears to be read before it is initialized. The Code Contracts solution is to wrap all such accesses to out-parameters in a call to the `Contract.ValueAtReturn` method. In our custom visualization, we omit these calls, but we make them explicit in Fig. 7a using a '↲' icon for better illustration. The same icon is used to indicate which parameters are out-parameters.



(a) (b)

Fig. 7. Wrapped references to the `min` and `max` out-parameters (a) and automatic wrapping/unwrapping during typing (b).

To address issue (B) above and avoid the inconvenience of calls to the `ValueAtReturn` method, Envision does not only omit them in visualizations but also inserts them automatically when the programmer types an out-parameter within a postcondition. Following concept 3 from Sec. II, we achieve this by adding a listener for expression modifications. If a method call to the `Ensures` method is edited, its arguments are "sanitized": all accesses to output arguments are wrapped, all other accesses are unwrapped. For instance, assume a programmer types `mini` within a postcondition of the `min_max` method as shown in Fig. 7b. When the programmer types the 'n' key, the symbol is resolved to an out-parameter and automatically wrapped. As soon as the second 'i' is typed, the symbol will be unwrapped as it no longer refers to an out-parameter.

V. Related Work

Modern development environments have advanced customizations that go beyond syntax highlighting using external plug-ins. For instance, the Code Contracts Editor Extensions for Visual Studio enhance a method declaration by showing inherited contracts and, for interfaces, contracts from contract classes using keywords. However, unlike our work, these visualizations are not editable—changes must still be made at the source location where contracts appear as normal method calls. The Editor Extensions also do not support the other customizations presented in this paper.

Davis and Kiczales [2] describe an approach to recognize syntactical patterns in code and visualize them with a more convenient notation. Our approach similarly allows the editor to choose an appropriate visualization for some code, but the decision is not just based on syntax and name bindings—the entire AST can be used together with the visual context and purpose. We also support customization of interactions.

Barista [3] is a structured editor framework that allows flexible presentation of code. Our prototype follows many of the same principles including flexible text-like input interactions, but also provides context-based customization, which is crucial to support eDSLs.

Intentional software [4] and MPS [5] are language workbenches that allow developers to create new textual and visual DSLs, integrated with a specially designed host language. To our knowledge neither tool supports the automatic context-sensitive visualization of program components as presented here, and unlike these tools, our approach works without introducing new languages or changing the host language.

Erwig and Meyer propose a framework for mixing textual and visual languages [6]. They use a text editor for the host language that allows language constructs to be created using graphical notations, particularly for specialized domains. However, unlike our work, their approach does not allow the user to simply customize the appearance of arbitrary language constructs. Moreover, parsing a program with mixed notations requires user-defined visual grammars.

VI. Conclusion

We proposed concepts of code editors that enable easy customization of the visualization and interaction for embedded DSLs and applied them to customize our structured editor Envision for .NET Code Contracts. Our results suggest that it is viable to extend editors with domain-specific customizations to aid programmers. Further work is needed to quantitatively evaluate the effectiveness of the proposed techniques.

References

[1] T. Kosar, M. Mernik, and J. Carver, "Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments," *ESE*, vol. 17, pp. 276–304, 2012.

[2] S. Davis and G. Kiczales, "Registration-based language abstractions," OOPSLA '10, pp. 754–773.

[3] A. J. Ko and B. A. Myers, "Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors," CHI '06, pp. 387–396.

[4] C. Simonyi, M. Christerson, and S. Clifford, "Intentional software," OOPSLA '06, pp. 451–464.

[5] M. Fowler. (2005, June) A language workbench in action - mps. [Online] Available: http://martinfowler.com/articles/mpsAgree.html.

[6] M. Erwig and B. Meyer, "Heterogeneous visual languages-integrating visual and textual programming," VL '95, pp. 318–325.