# Management of Guided and Unguided Code Generator Customizations by Using a Symbol Table

Pedram Mir Seyed Nazari  Alexander Roth  Bernhard Rumpe

Software Engineering
RWTH Aachen University, Germany
{nazari,roth,rumpe}@se-rwth.de

## Abstract

An essential part of model-driven development to systematically generate concrete source code from abstract input models are code generators. Regardless of their importance, abstract input models are not always suited to describe the output in a concise and precise way. Hence, customizations and adaptations of the code generator and the generated products are needed. Existing approaches mainly regard the code generation process as their primary concern and require the set up of an additional infrastructure in order to manage the customizations and adaptations. Thus, the goal of this paper is to propose an extension for template-based code generators to enable customizations and adaptations within a code generator that also respects referential integrity and reuses existing data structures for efficient management. First, we present a classification of common code generator customization and adaptation approaches (guided and unguided approaches) to identify the main concepts and elements of the approaches. Then, using the derived information relevant to manage guided and unguided approaches, we reuse the existing data structure (symbol table) to manage the customizations and adaptations. We achieve this by associating all relevant information directly with a template. This approach enables dynamic management of customizations and adaptations at run-time of the code generator and allow for statically checking before code generation. Our main contribution is an approach to combine guided and unguided customization approaches with a symbol table for efficient management.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

*DSM'15*, October 27, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3903-2/15/10...$15.00
http://dx.doi.org/10.1145/2846696.2846702

***Categories and Subject Descriptors*** D.2.11 [*Software Engineering*]: Software Architectures —Domain-specific architectures

***Keywords*** Code Generator Adaptation, Symbol Table, Template-based Code Generation

## 1. Introduction

In order to systematically generate source code in a model-driven development (MDD) environment, code generators are essential. In general, a generator is a software system that generates a concrete implementation from an abstract input model and consists of a *front-end* and a *back-end* [12]. While the front-end is concerned with language processing to create an *abstract syntax tree* and a *symbol table*, the back-end is responsible to systematically generate code from these abstract representations of the input model.

Even though a code generator is of high importance and full code generation is a goal of MDD, which should be achieved by providing multiple abstract models for each part of the generated system, abstract models are not always suited to describe the output in a concise and precise way. For instance, algorithms usually cannot be described in a more abstract and easier form than the implementation of those algorithms itself. Hence, it is an intrinsic property of a good code generator to be able adapt either the code generator itself or the generated product. An overview of existing approaches for managing variability and customizations of code generators [3, 12, 13] shows that their primary focus is on the whole code generation process. Customizations and adaptations are either handled before or after generation-time (i.e., run-time of the code generator), e.g. aspect-oriented code generation [8]. Within a code generator such concerns are barely addressed by e.g. Preprocessors [3] and aspect-oriented programming [14]. Hence, referential integrity, i.e., checking if the referenced customization exists, is not ensured.

Thus, in this paper we propose an approach to manage customizations and adaptations of code generators explicitly, statically, and dynamically at generation-time by employing a symbol table. Note that we explicitly neglect customizations of the input model and only regard customizations and adaptations within the code generator, because further research is required to address related challenges, e.g. *How can modular languages be managed in code generators? How can language customizations be handled by the code generator?*. We first restrict existing guided and unguided approaches to their basic elements in order provide required extensions to template languages. Guided customization approaches are approaches that explicitly define hook points that can be extended, but they do not allow for further customizations except for the hook points. In contrast, unguided approaches do not restrict customizations and adaptations. However, with this freedom of customization a high probability of failure is introduced. Having such an understanding of customizations, we use the symbol table as a data structure to explicitly and efficiently manage related information. For guided approaches, each hook point is stored in the symbol table. For unguided approaches, template replacements are stored. During generation-time the code generator can dynamically access this information. Whenever a hook point or template changes the changes directly affect the customizations. Moreover, after language processing, referential integrity can be ensured. In other words, the defined and configured customizations can be statically checked.

Hence, we first provide an understanding of guided and unguided customization approaches (Sect. 2) in order to extract their basic elements. Then, we identify static and dynamic checking as a major requirement in Sect. 3. Afterwards, we elaborate on how to use the symbol table for efficient management of guided and unguided customizations in Sect. 4. Finally, we concluded our paper in Sect. 5.

## 2. Guided and Unguided Customization of Code Generators

In a model-driven environment, where code generators generate source code artifacts, adapting a generated output means that the changes only affect one single artifact. Instead, adaptations of the code generator may affect all generated artifacts. For template-based code generators, adaptations mean changes to the templates. Adding functionality by template adaptations can, however, be challenging because the developer requires knowledge of the template, the languages (target and the generator language), and the architectures (generator and generated artifact architecture).

We classify customization and adaptation approaches of a code generator into two categories based on the overview of common approaches [5]. This classification aims at giving an understanding of the main characteristics of common customization and adaptation approaches. Completeness of categorizing all available approaches and their characteristics is not targeted. *Guided* approaches focus on explicit declaration of spots that can be extended. At the same time guided approaches explicitly forbid all other ways of customizations. For instance, variation points [2, 11] are a guided way that are defined at the design time of a code generator and later used for customizations. In the remainder of this paper, we refer to template languages that provide these extensions as *extended template languages*.

A major disadvantage is that during the evolution of a code generator variation points may change. As a consequence, the overall code generator needs to be adapted. This challenge can be seen as a variant of to the "fragile base class problem" [9]. In contrast, *unguided* approaches are less restrictive but are more error prone, as they allow to change every piece of the code generator and the generated code without explicitly denoting the elements that can be customized.

Subsequently, we elaborate on each type of customizations and present an abstract description of their realization.

### 2.1 Guided Customization

To avoid the complex and tedious task of adapting templates directly, *hook points* can be defined in templates. A hook point provides an approach to extend a template at a predefined spot. It is a place in a template that is planned for adaptation. Typically, such hook points are set during design time of a code generator and may be changed during the evolution of the code generator. Existing template languages provide such concepts, e.g. Xpand [4].

The basic elements of guided approaches are a way to define a hook point, and a way to bind values to a hook point. For template-based code generators each template may define multiple hook points, which are identified by a unique qualified name that consists of the path to the template, i.e., package name, the template name, and the hook point name. During configuration of the code generator hook points are bound to either one or multiple values. Each value can either be a simple string or another template. The values - in the case of a template it is the result of evaluating the template - are inserted at the spot the hook point is defined.

The major benefits of this approach are the guided way of customization, which shrinks the probability of errors, and the need to only regard a hook point not the overall code generator architecture. However, the lack of customizability is its main disadvantage, because each hook point needs to be carefully preplanned. Moreover, the result of a hook point may be syntactically invalid, since each template typically produces a string that conforms to a non-terminal of the target language.

### 2.2 Unguided Customization

Unguided approaches provide more freedom in adapting a code generator. The simplest approach is to directly adapt a code generator by e.g. adapting a template. Since templates build a complex structure, modifying them is hard. It requires a solid understanding of the template architecture and the template itself. This may also be challenging since the template sources may not be at hand.

Based on the knowledge of only the architecture, the basic concept of the approaches are the overriding of elements of the architecture. In the case of a template-based code generator, these elements are templates. The overridden template will not be used in the entire generation process anymore. A similar approach is provided by object-oriented programming languages in the form of overriding classes in generated code. An extension to template overriding is to add a new template before or after an existing template. This is similar to aspect-oriented programming [6]. Each time the existing template is executed, the added templates are either executed before or after.

Another extension that can be found for template-based code generators are global variables. Once a global variable is defined, it can be accessed from any template of the code generator. Such a value is a string and is used to exchange information between templates, e.g. names.

Certainly guided and unguided approaches do have commonalities. For instance, extending a template at the beginning of its execution can be seen as a generic hook point at each template start. The same holds true for extensions at the end of a template execution. Overriding a template is similar to point at the call of a template. It can also mimic extension by defining a new template, add an explicit hook point at the beginning and the end, and call the original template.

The flexibility introduced by this approach is dangerous as no type check ensures correctness of the resulting code. Only compilers will detect syntax errors. Clearly, code generators cannot prevent or detect such errors, but such customizations increase the potential of syntax errors drastically, as every adaptation is possible. Additionally, when overriding templates, a template may be in use elsewhere. This may cause side effects.

## 3. Requirements for a Data Structure to Manage Customizations and Challenges

Even though guided and unguided customization approaches for template-based code generators introduce adaptability, they also come with several challenges, which we subsequently explain in greater detail with the help of an example.

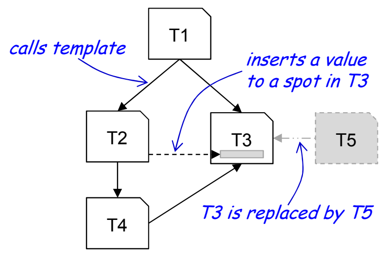

**Figure 1.** Example of a template architecture with template replacements and hook points.

Assuming the template architecture as shown in Fig. 1 to be given. When a code generator realizing this template architecture is started, the root template `T1` is called. This template then calls two other templates (`T2` and `T3`). Finally, `T2` calls `T4`, which then calls `T3` again. Furthermore, we assume that `T2` adds some content to a hook point defined in `T3` but `T3` is replaced by `T5` at generation-time, i.e., runtime of the code generator. In this situation two challenges are present.

The first challenge when realizing guided and unguided customizations in template-based code generators is static checking. For example, when a hook point is defined within a template and a value is bound, the hook point's name may change. In order to prevent resulting compilation errors of the generated code and make this error visible as soon as possible static checking of hook points and template replacements needs to be addressed. With existing mechanisms of current template languages, this is not feasible. To realize static checking an additional infrastructure is required.

The second challenge is to manage the template replacements and hook points dynamically at generation-time. In particular, while this information is defined statically, the values are accessed dynamically and may change during execution of the code generator. As most template languages provide global variables, they can be used to realize such behavior. A major disadvantage of this approach is that global variables are managed locally by one data structure and, thus, the information is not stored where it belongs to, i.e., at the template. Consequently, a mapping of template to dynamic information (template replacement and hook point values) is required.

In the remainder of this section, we identify further requirements for a data structure to efficiently manage customizations.

### 3.1 Data Structure Requirements

In order to efficiently manage and manipulate information related to customizations, an adequate structure is needed. In the following, we specify the requirements for a such a structure by the example of two simplified templates.

```
${defineHP("HP1")}
${defineHP("HP2")}
${setGV("aGV", "aVal")}
...
```

**Listing 1.** Template `p.T1` contains two hook point definitions and a global variable.

```
${defineHP("HP1")}
${bindHPString("p.T1.HP1", "value of hp"}
${replace("q.T3", "p.T1")}
...
```

**Listing 2.** Template `k.T2` defines a hook point, binds a string value, and replaces a template.

#### Requirements

Lst. 1 and Lst. 2 demonstrate some syntactical usages of the concepts introduced in Sect. 2. The first two statements of template T1 in Lst. 1 define the hook points HP1 and HP2. These two hook points are *related to* T1, but are bound by other templates. Thus, the first requirement is to **manage template specific information that is defined *within* that template and make it accessible (from outside) *(RQ1)*.**

Furthermore, information specific to a certain template can also be defined within other templates. The replace statement in Lst. 2 (l. 3), for example, replaces the template q.T3 (not shown here) by p.T1. Although, syntactically contained in k.T2, it is important to associate this information with q.T3 instead of k.T2. The reason is that a template that uses (e.g., includes) q.T3 might not know about k.T2. Hence, it must obtain the information from q.T3. Following from this, the second requirement is to **manage template specific information that is defined *outside* that template and make it accessible *(RQ2)*.**

The last statement in Lst. 1 defines the global variable aGV with the string value "aVal". This variable can be accessed and manipulated *by any template*. Other than the previous cases, it is *not* associated with a specific template. Hence, the third requirement is to **manage global values, i.e., template *unspecific* information, that can be defined from within *any template* and make them accessible *(RQ3)*.**

The requirements *RQ1*, *RQ2*, and *RQ3* concern the *definition* of information, i.e., definition of hook points, replacements, and global variables. However, these information must also be retrieved. The bindHPString statement in Lst. 2 (l. 2), for instance, *refers to* the hook point HP1 that is defined in p.T1. Analogously, the replace statement (l. 2) *refers to* the templates q.T3 and q.T1 in order to define a new replacement. From this, the fourth requirement follows: **given a reference, the corresponding definition must be obtained in order to access its associated information *(RQ4)*.** The definition can be in the same (intra-model) or in another template (inter-model) as the referencing template *(RQ4.1)*.

## 4. Symbol Table for Templates

A natural choice to tackle the challenges and requirements described in Sect. 3 is a *symbol table* which is a data structure that maps names to essential model elements [1]. A symbol table entry is called *symbol* (resp. symbol definition), which contains all essential information about a model (element) and has a specific *kind* depending on the model (element) it denotes. The symbol table is built up during the analysis phases [1].

In the extended template language as described in Sect. 2, essential model elements are, among others, templates and hook points. Hence, the symbol table can store corresponding template and hook point symbols including their associated information (fulfills ***RQ1*** and ***RQ2***). A symbol table can also manage built-in types that are accessible by all elements [10]. This concept can be used for global variables described in Sect. 2.2 (fulfills ***RQ3***). Furthermore, given a name and a (symbol) kind, the symbol table's resolving mechanism finds the corresponding symbol definition with all its associated information (fulfills ***RQ4***). The resolving begins within the referencing model and—if not found—it continues with other models (fulfills ***RQ4.1***). Furthermore, symbol tables usually provide some more concepts, such as visibility and import mechanisms [7].

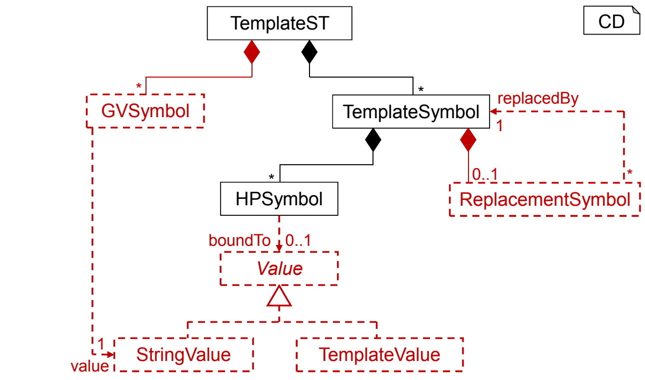

**Figure 2.** Symbol table structure for templates.

Fig. 2 demonstrates a symbol table infrastructure for the extended template language. TemplateST is the root of the symbol table and, thus, contains directly or indirectly all other elements (resp. symbols). A template is represented by TemplateSymbol. Essential elements of a template are hook points and replacements. Consequently, dedicated symbols for each of them exist, namely HPSymbol and ReplacementSymbol, which are stored in the TemplateSymbol. A HPSymbol can be bound to a Value, which is either a plain string (represented by StringValue) or a template (represented by TemplateValue). Note Value and its subclasses are no symbols themselves, but data associated with HPSymbols and GVSymbols. A TemplateSymbol T optionally has a ReplacementSymbol stating by which other template T is replaced. GVSymbols represent global variables. Since template-unspecific, they are directly stored in the TemplateST instead of in a TemplateSymbol. A global variable has a StringValue.

### 4.1 Example

Fig. 3 shows an instance of the symbol table for the templates specified in Lst. 1 and Lst. 2. The root element of the symbol table is an instance of TemplateST. T1:Template Symbol represents the p.T1 template (see Lst. 1). The two hook points HP1 and HP2 (ll. 1-2) are represented by instances of HPSymbol. Since defined *within* p.T1, T1:TemplateSymbol contains these two. The global variable aGV is defined in p.T1, but directly stored in :TemplateST, to be visible for every symbol.

Similarly, for the template k.T2 (see Lst. 2) a corresponding TemplateSymbol exists and is stored in :TemplateST. Also, HP1:HPSymbol is defined in T2:TemplateSymbol

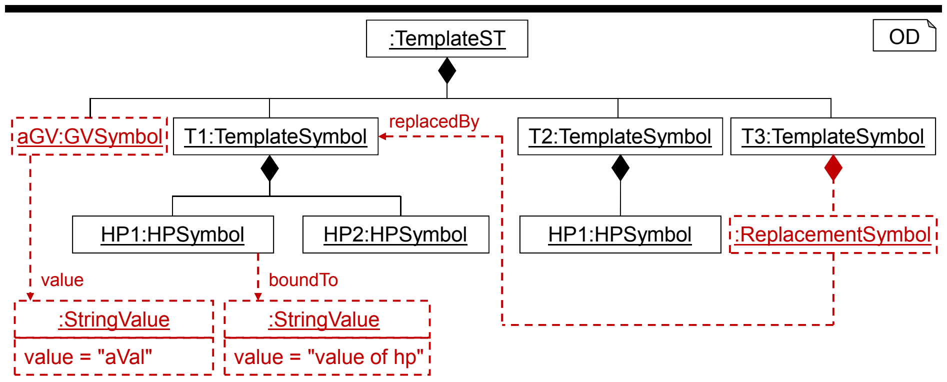

**Figure 3.** Symbol table instance for Lst. 1 and Lst. 2.

(l. 1). In line 2 of Lst. 2, the `HP1` hook point of `T1` is bound to the string `"value of hp"`. This information concerns `HP1`, hence, the corresponding symbol in `T1:TemplateSymbol` stores this information, highlighted by a `StringValue` instance linked with `HP1:HPSymbol`. That way, `HP1:HPSymbol` manages an information that is associated with it, but defined *outside* its own template. Similarly, the replacement in line 3 is associated with the template `q.T3`, and thus, a `ReplacementSymbol` is stored in the `T3:TemplateSymbol`, not in `T2:TemplateSymbol`. Additionally, the replacement symbol has a reference to `T1:TemplateSymbol` to indicate that `q.T3` is replaced by `p.T1`. The link is resolved by the symbol table's resolving mechanism.

### 4.2 Static vs. Dynamic Information in the Symbol Table

The symbol table in Fig. 3 contains two different types of information: static and dynamic information. *Static information* can be obtained without executing the templates, e.g., during design-time, whereas *dynamic information* can only be determined at generation-time. Hence, the static information is extended during the runtime. In Fig. 2 and Fig. 3 dynamic elements are highlighted with dashed lines.

***Static Information*** The templates as a whole are stored in corresponding files. From this files the name of the template can be obtained, e.g., the template `p.T1` is stored in `p/T1.tmpl`. Furthermore, all template-specific information defined within that template are static. This applies to hook points. Consequently, by solely analyzing the template files, we can build up a symbol table containing all `TemplateSymbols` and their `HPSymbols`. This enables conducting static validations in order to check the referential integrity of the templates. Template `k.T2`, for example, refers to the hook point `p.T1.HP1` (l. 2, Lst. 2) and the templates `q.T3` and `p.T1` (l. 3). The static information in the symbol table enables to check whether the hook point and the two templates exist. If, for instance, `q.T3` does not exist, it will be detected before code generation.

***Dynamic Information*** A global variable, is considered to be a dynamic information. The reason is that the template it is defined in might not be processed during runtime. Hence, storing the global variable in the symbol table during design-time might be wrong. For example, `p.T1` defines the global variable `aGV` (l. 3, Lst. 1). However, this has no affect if `p.T1` is not executed itself. Also, `aGV`'s value can be changed from within any template. Similarly, the `boundTo` information of a hook point can be set from any template. In Lst. 2 (l. 2), the template `k.T2` binds the hook point `HP1` of `p.T1` to the string value `"value of hp"` and overwrites the previous binding, if one existed. Finally, the replacement information can be (re-)set from within any template, and therefore, is a dynamic information, too.

## 5. Conclusion

While code generators are an integral part of model-driven development, customization approaches for the code generator itself that allow for dynamic and static checking for template-based code generators are currently lacking.

Hence, in this paper we extracted the basic elements of existing customization approaches and provide a classification and the basic elements of guided and unguided customization approaches. For guided approaches hook point definition and binding a value to a hook point are essential. For unguided approaches template replacements are the basic concepts. With this understanding, we derive requirements that have to be fulfilled for a data structure that should manage the customizations. A data structure that naturally fulfills these requirements is the symbol table. We have employed the symbol table to manage guided and unguided customization. The approach shows that static and dynamic checking of customizations become feasible. In consequence, errors can be detected before execution of the code generation rather than afterwards.

## References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley series in computer science. Pearson Addison-Wesley, 2007.

[2] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., February 2012. ISBN 0-201-70332-7.

[3] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[4] Eclipse. Xpand / xtend reference, 2012. URL `https://eclipse.org/modeling/m2t/?project=xpand`.

[5] T. Greifenberg, K. Hölldobler, C. Kolassa, M. Look, P. Mir Seyed Nazari, K. Müller, A. Navarro Perez, D. Plotnikov, D. Reiss, A. Roth, B. Rumpe, M. Schindler, and A. Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In S. Hammoudi, L. F. Pires, P. Desfray, and J. F. Filipe, editors, *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, pages 74–85, Angers, Loire Valley, France, February 2015. INSTICC and ESEO, SciTePress.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akit and S. Matsuoka, editors, *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-63089-0.

[7] H. Krahn, B. Rumpe, and S. Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *Software Tools for Technology Transfer (STTT)*, 2010.

[8] A. Mehmood and D. N. A. Jawawi. Aspect-oriented model-driven code generation: A systematic mapping study. *Inf. Softw. Technol.*, 55(2):395–411, 2013. ISSN 0950-5849.

[9] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, pages 355–382, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-64737-6. URL `http://dl.acm.org/citation.cfm?id=646155.679700`.

[10] T. Parr. *Language Implementation Patterns: Create Your Own Domain-specific and General Programming Languages*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2010.

[11] K. Pohl, G. Böckle, and F. J. v. d. L. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., 2005. ISBN 3540243720.

[12] A. Roth and B. Rumpe. Towards Product Lining Model-Driven Development Code Generators. In *MODELSWARD 2015: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, Angers, France, 9-11 February 2015. SciTePress.

[13] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[14] M. Völter. OpenArchitectureWare 4.1 Using AOP in templates. http://www.openarchitectureware.org/ pub/documentation/4.1/35_templateAOP.pdf, 2006.