Retrofitting Type-safe Interfaces into Template-based Code Generators

Kai Adam, Arvid Butting, Oliver Kautz, Jerome Pfeiffer, Bernhard Rumpe and Andreas Wortmann
Software Engineering, RWTH Aachen University, Aachen, Germany

Keywords: Model-driven Development, Template-based Code Generation.

Abstract: Model-driven development leverages transformations to produce general-purpose programming language artifacts. Model-to-text (M2T) transformations facilitate ad-hoc transformation development by requiring less preparation than model-to-model transformations. Employing template engines is common for M2T transformations. However, the M2T transformation artifacts (templates) rarely provide interfaces to support their black-box integration. Instead, composing templates requires in-depth expertise of their internals to identify and pass the required arguments. This complicates their reuse, and, hence, code generator development. Where switching to more expressive template engines is not feasible, conceiving templates as models can alleviate these challenges. We present a method to retrofit type-safe signatures into templates, generate typed interfaces from these, and show how this can be utilized to compose independently developed templates for more efficient code generator engineering.

1 INTRODUCTION

Model-driven development (France and Rumpe, 2007) lifts abstract models to primary development artifacts. These are better suited to automated analysis and transformation. For the transformation of models, two paradigms have emerged: model-to-model (M2M) and model-to-text (M2T). The former employ transformation languages, such as ATL (Jouault et al., 2006), to translate models of a source language into models of a target language, which requires having a formalization of the target language at hand. The latter translate models of a source language into arbitrary textual artifacts. Usually, such M2T code generators read models of the source language and either use string concatenation or templates to produce general-purpose programming language (GPL) artifacts. Templates encode transformations in text resembling the target GPL augmented with static and dynamic template language constructs such as conditionals or loops. Parametrized template calls assign the passed arguments to the templates' variable elements (*e.g.,* variables). In case the parameters expected by templates are not made explicit – such as with FreeMarker (Forsythe, 2013) or Velocity (Harrop, 2004) – developers have to understand the internals of the templates to be (re)used and identify their parameters manually. This requires cumbersome white-box template inspection. Where switching to a code generator employing a template engine supporting type-safe template integration is not possible due to legacy constraints, *retrofitting* it into existing generators can facilitate their development.

We present a concept to retrofit type-safe template integration into template-based code generators. To this end, we conceive templates as models, integrate signatures non-invasively, and generate GPL interfaces that enable type-safe template calls. As most template engines support using GPL APIs, the latter especially enables type-safe template integration from other templates and facilitates reuse of templates by enabling their type-safe black-box integration. The contribution of this paper, hence, is:

- A concept for retrofitting type-safe template integration into existing code generators.
- The application of this concept using the FreeMarker template engine.
- A case study on enhancing the code generators of the MontiArcAutomaton architecture modeling infrastructure.

In the following, Section 2 describes preliminaries, before Section 3 motivates the benefits of type-safe template integration. Afterwards, Section 4 introduces our concept and Section 5 its implementation. Subsequently, Section 6 describes the case study. Section 7 discusses related work and Section 8 debates observations. Ultimately, Section 9 concludes.

Adam, K., Butting, A., Kautz, O., Pfeiffer, J., Rumpe, B. and Wortmann, A.
Retrofitting Type-safe Interfaces into Template-based Code Generators.
DOI: 10.5220/0006605001790190
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 179-190
ISBN: 978-989-758-283-7

2 PRELIMINARIES

We demonstrate our concept using the FreeMarker template engine. Therefore, we enrich templates with signatures, parse these templates with the MontiCore language workbench, and produce Java classes wrapping the templates according to their signatures. MontiArcAutomaton is used as a running example throughout the paper to demonstrate the benefits of retrofitting type-safe template integration into existing code generators. This section introduces FreeMarker, MontiCore, and MontiArcAutomaton.

2.1 FreeMarker

The FreeMarker template engine (Forsythe, 2013) supports features typical to template languages, such as static text, comments, variables, control structures, various built-in operations, and template calls. Passing arguments to templates is implicit with FreeMarker: the invoked template inherits all variables from the invoking template. This lack of explicit parameters prevents template developers from identifying the template's contract, consisting of the parameters required to perform as desired, the returned values (if any), and side effects (*e.g.,* defining new variables). Consequently, errors in parametrization (missing arguments, wrong types) and issues caused by side effects that introduce dependencies (such as setting variables expected by other templates) are detected at template run time earliest.

2.2 MontiCore

MontiCore (Haber et al., 2015) is a workbench for efficient modeling language development. It employs EBNF-like, context-free grammars (CFGs) for integrated definition of concrete and abstract syntax. To validate constraints not expressible with CFGs, MontiCore features a compositional context condition framework (Völkel, 2011), where context conditions are well-formedness rules reified in Java. From a language's CFG, MontiCore generates the corresponding abstract syntax classes and an infrastructure to parse textual models into abstract syntax tree (AST) instances. The AST instances store the content of models, free from concrete syntax. For code generation, MontiCore also features a template-based code generation framework based on FreeMarker to translate AST instances into arbitrary target representations. Moreover, MontiCore supports compositional language integration (Clark et al., 2015) via inheritance, embedding, and aggregation (Haber et al., 2015).

```
01 component PathEvaluator[int min] {                    MAA
02   port in int dist,
03   port out boolean pathBlocked;
04                                    directed ports to
05   behavior automaton {              the environment
06     states f, b;
07     initial f;
08     f -> f [dist >= min] / pathBlocked = false;
09     f -> b [dist < min]  / pathBlocked = true;
10     b -> b [dist < min]  / pathBlocked = true;
11     b -> f [dist >= min] / pathBlocked = false;
12   }                          embedded automaton language
13 }
```

Figure 1: Atomic component embedding an automaton model of another stand-alone language.

2.3 MontiArcAutomaton

MontiArcAutomaton (Ringert et al., 2015) is an architecture modeling infrastructure realized with MontiCore. It comprises a component & connector architecture description language (ADL) (Medvidovic and Taylor, 2000) and several code generators translating models into various GPL realizations (Ringert et al., 2014). Its ADL leverages MontiCore's language embedding features to integrate modeling languages into components that describe their behavior.

MontiArcAutomaton distinguishes component types (denoted components) and component instances (denoted subcomponents), which are the units of computation and define interfaces of typed, directed ports. Composed components yield configurations of subcomponents that exchange messages via connectors between their interfaces. Atomic components instead embed a behavior description of a suitable modeling language. A corresponding model of an atomic component is depicted in Figure 1. The component of name PathEvaluator is used to determine whether there are obstacles in front of a mobile robot. To this end yields the configuration parameter min as a threshold (l. 1) and an interface of two ports (ll. 2-3). Its behavior is governed by an automaton model (ll. 6-11), which is embedded from another modeling language.

MontiArcAutomaton employs the parsers and AST classes generated by MontiCore from its CFG to translate textual architecture models into AST instances. It uses MontiCore's template-based code generation framework to translate the ASTs into GPL artifacts. To support embedding different behavior languages into MontiArcAutomaton components, being able to efficiently combine their code generators is obligatory. Consequently, understanding the contract of their templates is crucial.

3 EXAMPLE

Consider developing a code generator translating MontiArcAutomaton component models to executable Java artifacts using FreeMarker. With different component behavior languages available, this also requires integration of their respective code generators. Obviously, development begins with implementing a template for transforming each component model to Java. An excerpt of the template responsible for producing a class from the component model is depicted in Figure 2.

```
01 public class ${ast.name} {                                    FM
02   <#list ast.ports as port>
03     <#include "cd/Attribute.ftl">      access to
04   </#list>                             model under
05   <#list ast.subcomponents as sc>      transformation
06     private ${sc.type.name} ${sc.name}(${sc.params});
07   </#list>
08                                        template
09   public void compute() {              composition
10     <#if ast.behavior.isAutomaton>
11       <#include "automaton/Main.ftl">
12     <#else>
13       <#list ast.subcomponents as sc>
14         this.${sc.name}.compute()
15       </#list>
16     </#if>                             FreeMarker
17   }                                    constructs
18   // Additional transformation parts
19 }
```

Figure 2: Excerpt of a template transforming component models to Java. To this end, it includes the template Attribute (l. 3) for the creation of Java attributes.

This template produces a class of the same name as the component model (referencing to the model under transformation as `ast`, l. 1). Afterwards, it iterates of each port and produces a Java attribute by reusing the template Attribute from a generator translating class diagrams to Java (ll. 2-4). To this effect, the developer has to specify path and parameters of that template. As the latter are not part of the template call, she has to investigate the included template and identify which of its variable parts are expected as parameters. This may entail following the chain of included templates to identify all expected parameters. After creating members for each subcomponent (ll. 5-7), it produces a method that is invoked whenever the component should compute its behavior (ll. 9-17). In this method, the template distinguishes between embedded automata (l. 11) and composed components (ll. 13-15). In case of an embedded automaton, a corresponding template from the respective generator is included. This, again, requires comprehending its internals to pass the correct arguments. For composed components, their `compute()` method is called.

The included template `Attribute` is depicted in Figure 3. It does not explicate, which of its variables parts are expected as parameters (here `type` and

```
01 <assign instanceName = name.toLowerCase()>                    FM
02 private ${type.qualifiedName} ${instanceName};
03
04 public ${type.getFullName()} get${instanceName}() {
05   return ${instanceName};
06 }
```

Figure 3: Template creating attributes of a Java class.

`name`) other than by not defining these. Consequently, it also does not explicate their expected types. Whether `type` is actually a type-related object with various properties or just the type's name is not elaborated. To successfully include this template, the developer hence must understand its internals.

Consequently, the template integration depicted in Figure 2 will fail at run time as executing the Attribute template without type and name yields errors. Instead of switching to another, type-safe, template engine – which entails redeveloping all existing code generators – the developer desires to make template calls in to already employed engine type-safe and to report related errors as soon as possible. This raises the following challenges:

RQ1 The parameters expected by a template are explicit and statically typed.

RQ2 The side effects of template integration, such as defining new variables relevant to other templates, are made explicit.

RQ3 The compatibility of template arguments can be checked at design time.

RQ4 The resulting concept enables reusing existing templates without modifications.

The next section presents a concept that addresses these requirements via explicit template signatures and enables a more efficient reuse of templates.

4 TYPE-SAFE TEMPLATE INTEGRATION

This section presents a notion of template signatures and the process of translating these into a GPL code generator API able to invoke templates in a stable, black-box fashion while supporting reuse of existing templates. To this end, first the desired properties of template signatures are be defined and reified in the template language. Afterwards, these signatures are translated into GPL classes usable as strongly-typed template interfaces. Finally, this is retrofitted into existing code generators. The concept's activities are depicted in Figure 4, which is explained in the remainder of this section.

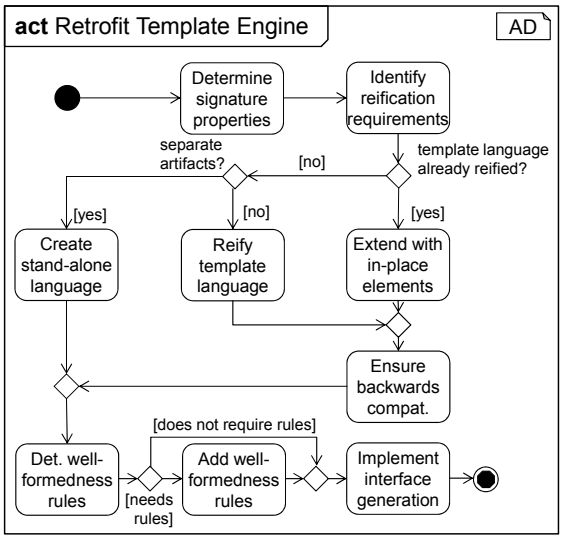

Figure 4: Retrofitting a template engine.

4.1 Defining Template Signatures

A *template signature* defines the contract of a template as typed inputs and outputs. A typed signature includes an ordered list of typed arguments and, optionally, an ordered list of typed return value parameters. With this, template signatures are similar to method signatures in common imperative or object-oriented GPLs: The unambiguous fully qualified method name of a method signature is analogous to the fully qualified name of the template that defines the signature. The typed and named method parameters of a method signature correspond to the typed and named parameters defined by a template signature. The return type of a method signature is the similar to a template signature's return values. Multiple return values in a template signature can be bundled, *e.g.,* to a complex return type that comprises all return types as members.

A template signature's arguments define all inputs required by the template (**RQ1**). Except for local variables, only these arguments may be used inside the template. The effect of a template call therefore only depends on the passed input arguments. Explicitly defining a template's return value parameters enables explicating the side effects of its execution (**RQ2**). This, for instance, can be new variables calculated for future use. We propose supporting multiple return values to facilitate returning multiple side effects.

4.2 Reifying Template Signatures

Retrofitting type-safe template signatures into existing template engines requires means to associate these signatures with the corresponding templates. This requires integrating signatures into the template language of choice. A key challenge in extending templates with signatures is that the template engine still must be able to process extended templates, hence there are three integration options: (1) creating a dedicated modeling language for out-place stored signatures similar to the header files in C/C++; (2) reifying the template language with the language workbench of choice; or, if this already exists, (3) extend the reified template language with signatures using the extension mechanism of the selected language workbench. As the former increases complexity in maintaining and evolving the two separate artifacts (templates and their signature models), we propose to integrate signature information into the template language. For this, its grammar (or metamodel and syntactic mapping) must be available. Where a grammar exists, rebuilding a template parser using grammar-based language workbenches, such as MontiCore (Krahn et al., 2010), Neverlang (Vacchi and Cazzola, 2015), Spoofax (Wachsmuth et al., 2014), or Xtext (Eysholdt and Behrens, 2010), is straightforward. Where the template language already was reified, changes to it might break existing tools (such as template analyses). Hence, in the latter two cases, integration also requires to ensure backwards compatibility of the templates. This, for instance, can be achieved by eliminating the signature information through a pre-processing model transformation prior to executing the templates. Afterwards, well-formedness rules need to be identified. This includes the uniqueness of template parameters, the availability of their types, *etc.* and can implemented using the realizing language workbenches' facilities. This can have the form of Java context conditions (MontiCore), OCL rules (Xtext), or a specific meta-language (Spoofax). After realizing and integrating these rules, templates can be parsed and their well-formedness can be checked. Ultimately, a generator translating templates into template interface classes is developed.

4.3 Generating Template Interfaces

The process of generating template interface classes from signatures retrofitted into the template language of choice is depicted in Figure 5: From a template augmented with signature, a corresponding parser (produced by the language workbench of choice) produces an instance of its abstract syntax (its AST). In case as stand-alone template signature language was created, the parser processes these models. Otherwise, it processes the templates. During this, context conditions check the validity of the templates with re-

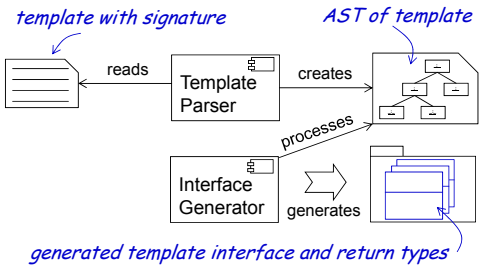

Figure 5: Generating template interfaces.

spect to the parameters. This includes checking that the name of each parameter and return types is unique and that the templates does not reference inexistent variables or parameters (**RQ3**). A template interface generator specific to the signature language (which can be the template language) and the GPL of template engine's API produces a GPL class representing the template's interface. This class yields methods to invoke the template with parameters and return types as defined by its signature. The generator used for producing the template interface can be different from the template engine to be wrapped. Hence, this also supports cross-engine code generation for template engines supporting the same GPL.

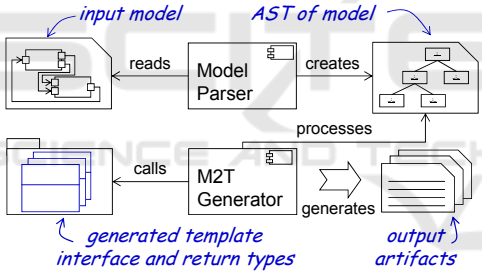

Figure 6: Using the generated interfaces.

This signature facilitates reusing provided templates via the generated interface classes as illustrated in Figure 6. Similarly, an input model of the language to process is translated into its AST representation. This representation is used by a M2T generator specific to the target language that now can rely on the generated template API to invoke templates in a type-safe fashion to produce the output artifacts as required. Assuming that the wrapped templates' engine supports leveraging GPL code (*e.g.,* for performing complex calculations during code generation), the generated template interfaces enables to use the same pattern for invoking templates from within other templates as well as from within GPL code artifacts. Not enforcing to use the generated GPL interfaces supports reusing templates employing the legacy template integration method (**RQ4**) where necessary, *i.e.,* its integration does not break existing code generators. Ultimately, the generated template interface class wraps a template and liberates template users from in-depth knowledge about its internal structure.

5 TYPE-SAFE TEMPLATE INTEGRATION FOR FREEMARKER

We present an implementation of our concept using FreeMarker, which leverages a MontiCore realization of the FreeMarker language that is extended with template signatures. Based on these, we produce template interfaces as explained above. The overall activities required to either develop new templates for type-safe interfaces or to retrofit existing templates are illustrated in Figure 7.

First, the generator developer has to decide whether she wants to create a new template from scratch or retrofit an existing template. For the former, she needs to determine the template's parameters and side effects. The side effects are translated into return values from which a template-specific return data type is generated. She specifies both in the template's signature and adds its body, which is performing the M2T transformation. In case the developer needs to retrofit an existing template, she has to comprehend the template under development to identify the parameters it expects and the side effects it causes (such as assigning values to other variables used by other templates). She specifies the side effects as return values and adjusts the template accordingly. After creating or retrofitting a template, it is parsed and its GPL interface class is generated. Additionally, in case the template defines not only required arguments but also provided return values, a result data structure capable of storing these is generated as well. The template interface class yields a single `generate(..)` method that takes the parameters specified in the template as input, starts template processing using the FreeMarker API, and returns an instance of the result data structure.

5.1 Integrating Template Signatures

Our concept relies on explicating the parameters and side effects of templates. To this effect, we reify the FreeMarker template language as a MontiCore language and extend it with template signature elements. We developed a MontiCore grammar for the FreeMarker template language and augmented it with optional template signatures. From this, MontiCore automatically generates a parser and abstract syntax classes for instances of the language. Our implementation

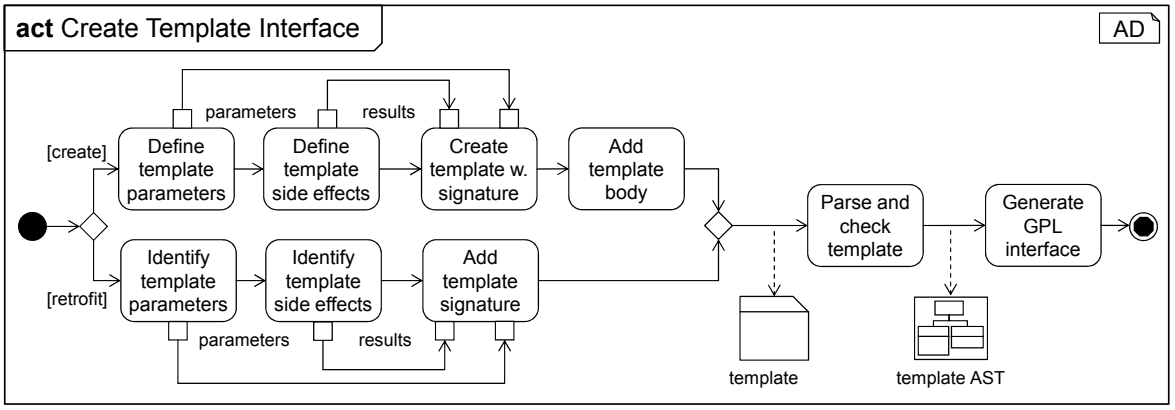

Figure 7: Overview of the generation process with statically typed template interfaces.

```
01 requires String name, Type type
02 provides String instanceName
03
04 <assign instanceName = name.toLowerCase()>
05 private ${type.qualifiedName} ${instanceName};
06
07 public ${type.getFullName()} get${instanceName}() {
08   return ${instanceName};
09 }
10
11 ${results.setInstanceName(instanceName)}
```

template signature

stores side effect instanceName in results data structure

Figure 8: Template generating an attribute implementation. Its signature defines parameters and return values.

executes this parser and checks for each parsed template whether its signature definition is correct (*i.e.,* no redundant names, referenced types exist, *etc.*) and valid in the template context (*i.e.,* no missing parameters). It also prevents multiple signature definitions in a template. If the template can be parsed and the context condition checks pass, the implementation continues with generating the template interfaces.

Regarding the data types of parameters and return types, we exploit that FreeMarker features a Java API and assume that the types are either built-in or handcrafted Java types and that they are referenced via their fully qualified names. Template developers define a template's signature directly at the beginning of the template. The signature comprises a list of typed and named parameters (**RQ1**) and, optionally, a list of typed and named return values (**RQ2**). As signatures are optional, this guarantees backward compatibility (**RQ4**) by construction. After parsing the template, this information is made explicit in the template AST.

Figure 8 depicts an improvement of the `Attribute` template of Figure 3: the template's new signature defines that it requires two parameters (l. 1) and provides a single return value (l. 2). The definition of parameters begins with the keyword `requires` and is followed by a list of input parameters consisting of the parameters' types and names. Similarly, the definition of returned values starts with the keyword `provides` and is followed by a list of typed and named return values. Both, required and provided parameters are optional. However, for generating a template interface class, at least required parameters (l. 1) must be provided. The subsequent list of provided return values (l. 2) is optional for this as well. In the template body, parameters are accessible by their names (*e.g.,* l. 4). A structure generated from the template's return values is accessible as `results` from within the template and can be used for explicating side effects (l. 11). Its methods and attributes are derived from the templates provided parameters. The interfaces generated for templates (a more detailed description is given in Section 5.2) enables passing arguments in a type-safe fashion and are usable from inside the template as well. Invoking the `generate()` method of an interface derived from a template yields an instance of its return value type that stores the text and all return values produced during template execution.

5.2 Generating Template Interfaces

The template interface generator implementation uses MontiCore's code generation framework, which itself employs FreeMarker. It takes an AST of a parsed template as input and produces a Java API (template interface) for template integration. The generated API enables type-safe template integration with respect to the arguments passed to a template and the return values produced as side effect during template execution. This ensures invoking templates in a type-safe fashion and, hence, facilitates their integration, and ultimately their reuse. For each retrofitted FreeMarker template the generator produces the following two artifacts:

1. A class of the template's name serving as the interface for type-safe template integration (*cf.* `JAttribute` of Figure 9),

2. A class wrapping the return values of the template's signature to capture its side effects (*cf.* the Java class `JAttributeResults` of Figure 9).

Generating and using these classes relies on other non-generated classes part of the run-time environment of our implementation. The quintessential types of our run-time environment are the Java interface `IResult` (bottom-left of Figure 9) and the Java class `GeneratorEngine`.

IResult. One feature all generated template interfaces have in common is that invoking their `generate()` method produce the text resulting from applying the wrapped template to the passed arguments. Consequently, all implementations of return value types must implement this interface, which specified that the return value type at least features the methods `setText(String)` and `getText()`.

Generator Engine. Each template interface uses the class `GeneratorEngine` (*cf.* Figure 9) as facade to the Java API of FreeMarker). The class provides a default generator setup, which enables to use the template interface without in-depth generator engine configuration knowledge. To this effect, it wraps all relevant generator configuration parameters. Nevertheless, in situations in which a generator configuration might be indispensable (*e.g.,* generator integration), the generator developer is able to modify the default configuration parameters by a smart generation gap. The `GeneratorEngine` class features a `process` method invoking the FreeMarker API to trigger type-safe template execution. The following focuses on the generation steps.

Based on the run-time environment types we produce two classes per template: return value type and its interface.

Return Value Type Generation. Our approach permits defining multiple return values in template signatures and aims at generating a high-level Java API in form of template interfaces. However, Java, for instance, only allows defining single method return types. Thus, to enable returning multiple return values in response to a template interface integration, for each template, the interface generator produces a class containing a field for storing the generated text and a field for each of the template's return values. Thus, an instance of the class returned after template integration contains the text generated by the template and the return values produced during template execution as side effects. For instance, the class `ComponentResults` is generated from the template `Component` that is illustrated in the top right part

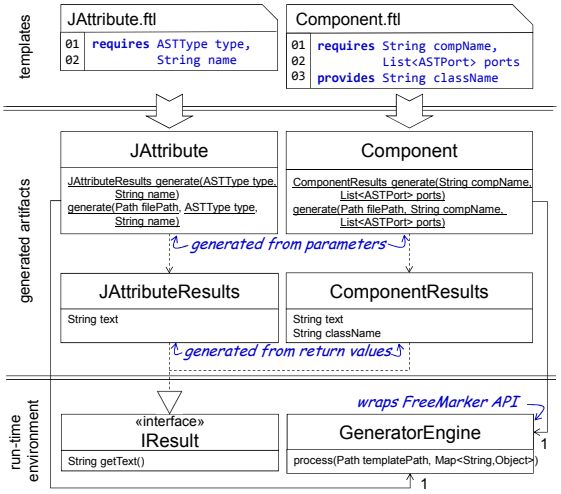

Figure 9: Different cases of template signatures and derived template interfaces.

of Figure 9. For each return value of the template's signature, the class yields a member of the same type and name as well as corresponding getters and setters.

In case a template does not define any return values, the generated class only contains a field for storing the generated text (*cf.* template `JAttribute` in Figure 9). Each generated class wrapping return values implements the interface `IResult` that provides the method `getText()` for retrieving the text produced during template execution. This facilitates generator integration via ordered GPL template calls and further processing of the generated text, as well as template integration via inline-template calls.

Template Interface Class Generation. For each template that contains a signature, the generator produces a class with the same fully qualified name as the template. This provides a static `generate()` method for template integration (Figure 9). For instance, the classes `Component` and `JAttribute` are generated of the respective same named templates. Each generated class is associated to the `GeneratorEngine` class, which is used by the generated method to wrap FreeMarker's Java API. For each of the template's parameters, the `generate()` method signature has a parameter of the same type and name. The order of the method's parameters is the same order as defined by the parameter list in the template signature. This facilitates type-safe template integration as inaccurate template interface method integration are detectable at design time (**RQ3**). The return type of the first generated method is given by the Java type generated from the template's return values. Besides the `generate()` method described above, the generator produces another `generate()` method that takes a

```
01 public ComponentResults generate(
02           String compName, List<ASTPort> ports) {
03
04  ComponentResults results = new ComponentResults();
05
06  HashMap<String, Object> params = new HashMap<>();
07  params.put("compName", compName);
08  params.put("ports", ports);
09  params.put("results", results");
10
11  String artifactAsString = this.generatorEngine
12                      .process(templatePath, params);
13
14  results.setText(artifactAsString);
15  return results;
16 }
```

Figure 10: Derived generate() method of the Component interface that returns an instance of ComponentResults.

file path as additional parameter. This method writes the text resulting from internally calling the first `generate()` method into a file located at the specified file path. While the first `generate()` method is mainly used for template and generator integration, the second `generate()` method is used for generating files.

The implementation of the first `generate()` method of the `Component` class is presented in detail in Figure 10. First, an object for wrapping the text resulting from template execution and the return values produced by side effects is instantiated (l. 4). Then, the attributes passed to the method are added to a `HashMap` instance `params` (ll. 6-8). As the return value wrapping object is added as additional explicit template execution parameter (l. 9), manipulating its content during template execution is possible. The map is passed to a (not type-safe) `process` method call on the `GeneratorEngine` (ll. 11-12), which takes care of removing template signatures from templates for backwards compatibility to the FreeMarker engine. Afterwards, the result from calling the template engine's API is stored in the designated field of the return value wrapping instance (l. 14), before the method returns it (l. 15). The generated method is invocable within templates and directly in Java artifacts.

The `process` method of the class `GeneratorEngine` expects as argument a map that assigns attribute names to `Object` instances. The instance assigned to an attribute name represents the attribute's value. As the Java type `Object` is a super type of any other Java type, no fine grained assurances (except for the assurances made by type `Object`) are given by the arguments. Thus, the `process()` method of the `GeneratorEngine` class (as well as the FreeMarker API) is not type safe as arbitrary Java type instances may be passed to it. However, as this method is invoked by a method generated from the template's signature, its type-safe use is entailed by construction.

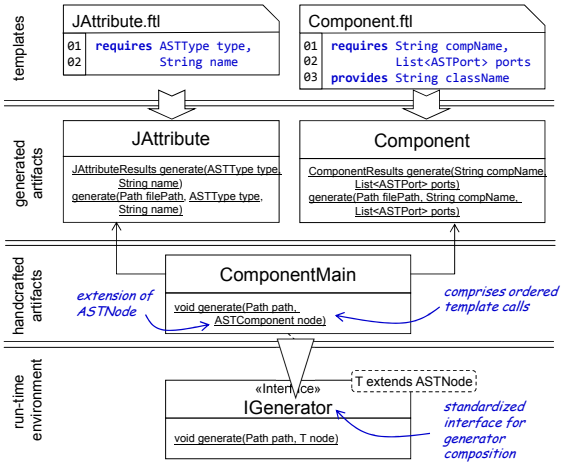

Figure 11: An overview of the artifacts generated and used for generator integration.

5.3 Using Type-safe Template Calls

Our approach guides generator developers to an easier and less error-prone development of template-based code generators by using generated template interfaces. Hence, an in-depth knowledge about the generator engine is not required.

Using Template Interfaces. Generator developers can either use the default or a custom generator engine configuration by a smart generation gap. In addition, the `generate()` methods of the template interfaces are statically typed. Consequently, faulty parametrization of template integration is recognized by the Java compiler at implementation time. For instance, if the generator developer invokes the attribute template (*cf.* Figure 8) via its template interface (*i.e.,* its method `generate(ports)`), the Java compiler recognizes incorrect parameters and causes an error. Furthermore, our approach generates an interface specific `Result` class that wraps the templates content and additional side effects as members.

It provides setters to assign values to the result parameters within the template for further processing. For instance, the `Attribute` template uses the setter method of this specific wrapper instance to assign the instance name of the artifact to the result `instanceName` (*cf.* l. 6 in Figure 8). Hence, the generator developer can access the result after M2T transformation and use it for further processing steps, such as generator and template integration.

Template Integration for Generators. Figure 11 illustrates the overview of the generated and used artifacts for generator integration. The `IGenerator` interface is part of our run-time environment and provi-

des a `generate()` method with parameters `path` and `node`. While the first parameter `path` enables the generator developer to set the file path, the second parameter `node` is a generic extension of the `ASTNode`, which must be specified in the generator implementation. In contrast to the generated template interfaces, generator integration requires a handcrafted generator implementation that implements the `IGenerator` interface. They enable template orchestration, such as manifesting template integration orders for various interfaces, employing transformations of template integration resulting return values, and passing those values into consequent template invocations.

For instance, the `ComponentMain` generator implements the `IGenerator` interface and enables the generator developer to employ this generator on instances of the `ASTComponent` type. An invocation of the `generate` method reuses the generated interfaces `JAttribute` and `Component` and hands specific `ASTComponent` substructures, such as a list of `ASTPort` instances, to the corresponding template interfaces. Ultimately, this enables reusing and composing template interfaces, as well as composing generators with a standardized generator interface. The next section presents a case study that introduces an application of generator integration using MontiArcAutomaton.

6 CASE STUDY

This section presents the integration of our type-safe template integration mechanism into the MontiArcAutomaton code generation framework. The aim of the integration is to facilitate integration of the MontiArcAutomaton structure code generator (responsible for translating ports, connectors, message passing, *etc.*) with the code generators responsible for translating models of embedded behavior languages. Prior to applying our retrofitting concept, the reusing FreeMarker-based code generators required invoking the templates responsible for translating embedded behavior models directly. This raised the aforementioned problems of having to comprehend template internals to pass the correct number and type of arguments expected by the respective templates.

The code generators of MontiArcAutomaton comprise 55 FreeMarker templates and 41 Java classes that are responsible for organizing template calls and calculations of template arguments outgoing from the passed component models. In our case study, the component structure generator employs two behavior implementation generators to generate behavior of components: one translates embedded Java/P (Schindler, 2012) programs, the other embedded I/O automata. Additionally, we reuse parts of a generator translating UML/P (Rumpe, 2016) class diagrams to Java. The quintessential templates of the MontiArcAutomaton structure generator and the I/O automaton generator are depicted in Figure 12. Each generator features a main template that defines its configuration (*e.g.,* assigning variables) and invokes other templates.

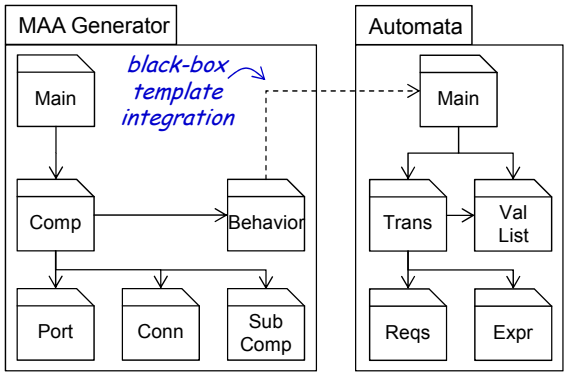

Figure 12: Behavior generator integration realization.

The MontiArcAutomaton component structure generator comprises a `Component` template that invokes templates transforming ports, connectors, subcomponents, and embedded behavior models. The latter invokes the main template of the integrated behavior generators. As neither the main template, nor the templates it invokes explicate the required parameters, the developer of the `Behavior` template must comprehend the internals of each template of the I/O automaton behavior generator. Similarly, the `Port` template of MontiArcAutomaton's structure generator uses the `Member` template of the class diagram generator, which entails the same challenges.

Each template of the MontiArcAutomaton code generation framework, the two behavior code generators, and the class diagram generator is post-hoc equipped with a signature. With this, all necessary template parameters are explicitly visible in the template's signatures. Therefore, our implementation generates a statically typed interface for the template calls, which facilitates the reuse of templates of other code generators. For example, the Java generator of MontiArcAutomaton generates private members with getters and setters for ports in generated component implementations. Similar to Java attributes, ports have a type and name.

With the generated template interfaces in place, reusing templates of the class diagram generator to produce class members from ports (passed as type and name) is straightforward and prevents producing redundant template code.

```
01 requires: ASTComponent comp
02 provides: String className, String behaviorKind
03
04 public class ${comp.name} {
05 ${results.setClassName(comp.name + "Impl")}
06 <#if comp.isAtomic>
07     ${results.setBehaviorKind(comp.behavior.kind)}
08 </#if>
09 <#list comp.ports as p>
10     ${JAttribute.generate(p.type,p.name).getText()}
11 </#list>                    type-safe template integration
12 }                            through generated interface class
```

Figure 13: Component template that returns the kind of the embedded behavior description if the component is atomic.

The correctness of the arguments passed to the templates is checked at design time, as the `generate()` methods of the template interface are statically typed with the type of the signatures' parameters. Furthermore, some of the MontiArcAutomaton templates additionally define results in their signatures. The results are used to perform post-generation steps that are closely related to or depend on the generated code. In MontiArcAutomaton, for instance, this is leveraged to realize generator integration of the `MAAGenerator` and the generators of employed behavior languages. The template, which generates a component implementation, returns the kind of the embedded behavior language (*cf.* l. 3 in Figure 13). MontiArcAutomaton then decides, which behavior generators should be called.

Black-box integration of generators relieves generator developers from knowledge about template internals and thus facilitates template reuse, which ultimately decreases the time required to implement code generators. As this use case example demonstrates, our concept supports to apply extension mechanisms from software language engineering to realize compositional, extensible code generators.

7 RELATED WORK

To the best of our knowledge, our concept to retrofit type-safe template integration into existing template engines is unique. The results in can achieve, however, are comparable to type-safety in existing template engines. This section discusses these relations and aggregates the results in Table 1, where the last column depicts the features of applying our template interface generation concept to FreeMarker as presented in Section 5.

Acceleo (WWWa, 2017) provides a M2T transformation language that supports developing transformations as plain text files containing transformation commands as well as target language text. Its templates yield a signature containing at least the abstract syntax concept being transformed. It supports an optional third parameter enabling to pass a list of arguments (of the same type). Hence, it supports type-safe template calls with the limitation of passing arguments of the same type or their most common base type only. It does, however, not support specifying the templates' return types.

The shortcomings of FreeMarker (Forsythe, 2013) are already discussed in Section 2.1. The Epsilon Generation Language (EGL) (Rose et al., 2008), Velocity (Harrop, 2004), and XPand (Klatt, 2007) are template engines similar to it. All support developing transformations as plain text files containing code generation commands between target language fragments. Velocity and XPand also rely on inheriting local variables to called, signature-less templates. EGL templates can be called from a template coordination DSL that can pass maps of arguments to the template. As the called template's required parameters are not made explicit, this does not allow for static type-safety checking.

The template engines Jamon (WWWb, 2017) and Twirl (Saxena, 2015) both support explicit signatures of template parameters and the generation of GPL template interfaces. Hence, both support statically typed template calls. Neither supports specifying the templates' return types, which complicates explicating the side effects of template integration.

Templates of the Java Emitter Templates (JET) (WWWc, 2017) feature a signature that consists of the qualified name of the Java class to be generated and a list of loosely typed parameters (Java `Objects`). While facilitating tracing between model (parts) and GPL artifacts, it does not support type-safe template integrations.

Xtend (Bettini, 2016) is a programming language extending Java with templating features that facilitate string concatenation. Using directives and control structures embedded into the template parts of a Xtend method supports producing the target language text using the methods' signature for the embedded template. While this returning complex structure to capture the side effects of template execution, the Xtend methods are not generated, hence they either require handcrafting the specific return types or rely on passing loosely typed structures (such as sets of Java's objects).

We captured the features of these template engines with respect to the capabilities of our retrofitting concept in Table 1. Fully configurable typed template parameters are supported by only half of the template engines and template return types are supported only by Xtend. However, none of the inspected template engines supports type-safe and complete specification

Table 1: Features of template engines compared to the results of applying our concept to FreeMarker (last column).

Feature	Acceleo	EGL	FreeMarker	Jamon	Twirl	JET	Velocity	XPand	Xtend	Retrofit
Typed template parameters	(✓)	×	×	✓	✓	✓	×	×	✓	✓
Template return types	×	×	×	×	×	×	×	×	✓	✓
Side effect specification by construction	×	×	×	×	×	×	×	×	×	✓
Statically typed template calls	×	×	×	✓	✓	×	×	×	✓	✓
GPL template interfaces	×	×	×	✓	✓	✓	×	×	✓	✓

of side effects by construction as proposed through generating result data structures from provided template parameters. While this can be emulated with Xtend by developing these data structures manually, for each template method, this requires greater effort and is more error-prone. However, where typed template parameters are supported, this usually employs GPL template interfaces that allow for statically typed template calls. Overall, this suggests that many existing template engines could benefit from retrofitting type-safe template integration into their languages and infrastructures.

8 DISCUSSION

Our approach to retrofit type-safe template integration into template engines exploits conceiving templates as models. It rests on making the parameters expected by templates explicit and deriving a strongly typed GPL API from these. Hence, each template's parameters are easily accessible and errors in their parametrization are detected by the compiler GPL at design time. This reduces the need to analyze the internals of the template to be (re-)used.

Although our concept reifies template signatures on GPL level and we strongly recommend to use the generated API even from within templates, we do not eliminate unsafe operations from the template engine. Considering, for instance, FreeMarker again, it is still possible to call templates using its built-in `include()` method and inheriting variables to the called template. Eliminating this could be achieved easily by enforcing a context condition prohibiting the related template language elements, but would break backward compatibility with existing templates.

Similar to M2M transformations, our approach supports to optionally specify return types of templates, *e.g.,* in terms of the abstract syntax classes produced by the respective template. This supports to combine it with M2M transformations by interleaving these with template calls and ultimately enables developers to choose the most suitable transformation paradigm as appropriate.

9 CONCLUSION

We presented a concept to support the development of template-based code generators by enabling a type-safe template integration that ultimately facilitates template reuse and hence code generator development. This concept relies on considering templates as models and reifying their template languages in a machine-processable fashion. Based on this, template interfaces are generated that support static checking of template argument compatibility, making side effects of template execution explicit, and enabling using the same template integration mechanism from GPL artifacts and from within other templates. We also illustrated the benefits of this concept using its FreeMarker-based MontiCore realization for improving code generators of the MontiArcAutomaton architecture modeling infrastructure. Despite strong requirements (especially reifying the employed template language), we believe that formalizing the interfaces of code generators can greatly facilitate their development and, ultimately, adoption of model-driven development.

REFERENCES

Bettini, L. (2016). *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd.

Clark, T., den Brand, M., Combemale, B., and Rumpe, B. (2015). Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, pages 7–20. Springer.

Eysholdt, M. and Behrens, H. (2010). Xtext: implement your language faster than the quick and dirty way.

In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 307–309, New York, NY, USA. ACM.

Forsythe, C. (2013). *Instant FreeMarker Starter*. Packt Publishing.

France, R. and Rumpe, B. (2007). Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)*, (2):37–54.

Haber, A., Look, M., Mir Seyed Nazari, P., Navarro Perez, A., Rumpe, B., Völkel, S., and Wortmann, A. (2015). Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, volume 580 of *Communications in Computer and Information Science*, pages 45–66. Springer.

Harrop, R. (2004). *Introducing Velocity*, pages 1–8. Apress, Berkeley, CA.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., and Valduriez, P. (2006). ATL: a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM.

Klatt, B. (2007). Xpand: A Closer Look at the model2text Transformation Language.

Krahn, H., Rumpe, B., and Völkel, S. (2010). MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372.

Medvidovic, N. and Taylor, R. (2000). A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*.

Ringert, J. O., Roth, A., Rumpe, B., and Wortmann, A. (2014). Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. In *Model-Driven Robot Software Engineering Workshop (MORSE'14)*, volume 1319 of *CEUR Workshop Proceedings*, pages 66 – 77.

Ringert, J. O., Roth, A., Rumpe, B., and Wortmann, A. (2015). Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57.

Rose, L. M., Paige, R. F., Kolovos, D. S., and Polack, F. A. (2008). The epsilon generation language. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 1–16. Springer.

Rumpe, B. (2016). *Modeling with UML: Language, Concepts, Methods*. Springer International.

Saxena, S. (2015). *Mastering Play Framework for Scala*. Packt Publishing Ltd.

Schindler, M. (2012). *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag.

Vacchi, E. and Cazzola, W. (2015). Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1–40.

Völkel, S. (2011). *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag.

Wachsmuth, G. H., Konat, G. D. P., and Visser, E. (2014). Language Design with the Spoofax Language Workbench. *IEEE Software*, 31(5):35–43.

WWWa (2017). Acceleo 3.1.0 user documentation. Accessed: 2017-06-08.

WWWb (2017). Jamon Java Template Engine - Jamon website http://www.jamon.org/. Accessed: 2017-05-03.

WWWc (2017). Java Ermitter Templates - JET Project website https://eclipse.org/modeling/ m2t/?project=jet. Accessed: 2017-05-12.