

Static Syntax Validation for Code Generation with String Templates

Dorian Weber^(✉) and Joachim Fischer

Institut für Informatik, Humboldt-Universität zu Berlin, Berlin, Germany
{weber,fischer}@informatik.hu-berlin.de

Abstract. Many applications of model-based techniques ultimately require a model-to-text transformation to make practical use of the information encoded in meta-model instances. This step requires a code generator that has to be validated in order to ensure that the translation doesn't alter the semantics of the model. Validation is often test-based, i.e. the code generator is executed on a wide range of inputs in order to verify the correctness of its output. Unfortunately, tests generally only prove the presence of errors, not their absence. This paper identifies the common core of string template implementations that are often used in the description of code generators, deriving a formal model that is suitable for mathematical reasoning. We provide a formal proof of the equivalence in expressiveness between string templates and context free grammars, thereby allowing the application of formal results from language theory. From there, we derive a scheme that would allow the verification of syntactical correctness for generated code before the translation of any model-instance is attempted, at the expense of freedom in the variability of the description.

Keywords: Code generation · Language theory · String template · Meta language · Domain-specific language

1 Introduction

The core tenet of model-driven engineering is the use of domain models to represent abstract knowledge about a particular application domain. These models are used to connect different problem sets through a shared information base which helps to reduce redundancy, increase modularity and facilitate the creation of less verbose programs through the use of domain-specific languages. In order to use a domain model for a particular problem set, it is often necessary to vary its representation, e.g. by applying a model-to-model or model-to-text transformation. The focus of this paper is on model-to-text transformations using string templates and it deals with the sub-problem on how to ensure the syntactical correctness of the generated text as a sentence of the output language. In other words, prior to seeing any meta-model instance under which circumstances can we guarantee that a string template will expand to syntactically correct code?

```

1  """
2      <body>
3          <h2>«module.name»</h2>
4          «FOR expr: module.member»
5              <h3>«expr.name»</h3>
6              <dl>
7                  «FOR it: expr.member»
8                      «IF it.universal»
9                          <dt>«it.name»</dt>
10                     <dd>«it.value»</dd>
11                     «ELSEIF it.extensible»
12                         <dt/>
13                         <dd><i>Enumeration is extensible</i></dd>
14                     «ENDIF»
15                 «ENDFOR»
16             </dl>
17         «ENDFOR»
18     </body>
19 """

```

Listing 1. *Xtend* string template for generating *HTML* code from an *ASN.1* meta-model instance.

The ability to guarantee the syntactical correctness for generated code has applicability in a wide range of related topics, e.g. in traditional compiler construction for specification-type languages (e.g. *ASN.1*), languages featuring syntax extension mechanisms (e.g. *SLX*), or languages that support string templates for code generation (e.g. *Xtend*). As a result, this area has been studied extensively with promising theoretical results but severe technological prerequisites. The goal of this paper is to provide a formal foundation for novel techniques addressing this problem with the expectation that the derived solutions will be simpler in terms of implementation and maintenance.

The paper is organized as follows. We begin by providing an instance of the problem we are attempting to solve, followed by taking a look at related work and discussing the relationship to this paper. We continue in Sect. 2 by formally defining string templates and providing a proof establishing their equivalence to context free grammars. Section 3 outlines a potential solution that would allow automatic proof-based verification for syntax of the generated language. Finally, Sect. 4 contains conclusion and outlook.

1.1 Brief Example

Listing 1 shows an example of a string template in *Xtend* that seems to generate *HTML* code. In line 10, we notice that the opening `<dd>` is closed by `</dt>`. Therefore, without knowing the concrete value of `module`, we deduce that the output can be invalid *HTML*. We can do so by analyzing the algorithm used to generate the code. Compare and contrast with listing 2 that seems to generate an enumeration in *C*. Here, the output looks like it should be syntactically valid,

but we cannot be sure since we don't know whether for example `expr.name` in line 2 will expand into a valid identifier or not.

Given the context free grammar for the syntax of a target language (e.g. *HTML*, *C*), this paper attempts to identify the circumstances under which it would be possible to decide whether all value configurations (i.e. meta-model instances) lead to syntactically correct code.

```

1  | """
2  |     typedef enum «expr.name» {
3  |         «FOR it: expr.member»
4  |             «IF it.univerval»
5  |                 «it.name» = «it.value»,
6  |             «ELSEIF it.extensible»
7  |                 /*
8  |                 * Enumeration is extensible
9  |                 */
10 |             «ENDIF»
11 |         «ENDFOR»
12 |     } e_«expr.name»;
13 | """

```

Listing 2. *Xtend* string template for generating an enumeration in *C* from an *ASN.1* meta-model instance.

1.2 Related Work

Parr discusses the relationship between string templates and context free languages in a semi-formal manner that includes an informal sketch for a proof [6]. Our paper provides formal answers for that topic.

Wachsmuth describes an algorithm to mechanically derive a *code template language* based on the grammar of an output language that guarantees syntax correctness [9]. While the paper contains a complete syntax and semantics description, no proof of correctness is offered. Dynamically evaluated expressions embedded in string templates fall outside of the scope of the paper as well.

Arnoldus deals with *syntax safe templates* in his PhD thesis [1], which constitute a language that is the result of augmenting a specific notation for string templates with the grammar of the target language, providing a proof-of-concept for the aforementioned paper by Wachsmuth. Using a parser for that grammar, the static parts of an interconnected set of string templates can be verified to be syntactically correct fragments of the output language. In order to guarantee full syntax correctness, one must also ensure that dynamic expressions embedded within the string template are validated, for which the author proposes a runtime scheme. This step changes the validation from proof to test based. The advantages are tangible nonetheless, since the static parts of string templates can be verified statically. The author makes no attempt to identify common features with other kinds of string template languages beyond his own. Our paper attempts to derive more general truths about this issue, as well as proposing a mechanism to capture dynamic expressions in the automated proof.

2 Relationship between Context Free Grammars and String Templates

In this section, we provide a formal proof showing that string template languages are alternative notations of context free grammars. This allows us to conclude that the general problem of deciding whether a set of string templates generate a subset of a context free language is undecidable. We begin by defining mathematical structures for the representation of context free grammars and string templates, outlining the latter's connection to the string template notation featured in *Xtend* as a representation of string template notations used in industrial strength languages. We continue with a constructive proof that maps context free grammars to string templates and vice versa while preserving the generated language. Finally, we discuss the consequences of this result.

2.1 Basic Definitions

Definition 1. A Context Free Grammar (CFG) is defined by the tuple (V, Σ, P, V_0) where

- V is a finite set of meta characters,
- Σ is a finite set of symbols, disjoint from V ,
- $P \subseteq V \times (\Sigma \cup V)^*$ is a finite relation,
- $V_0 \in V$ is the start symbol.

We denote the production rule $(S, \alpha) \in P$ as $S \rightarrow \alpha$.

Definition 2. A Context Free Language (CFL) is the set of all strings that can be produced by a CFG through application of a sequence of production rules via substitution of a meta character by the rule's right-hand-side.

For $\mu, \nu \in (\Sigma \cup V)^*$ we write that $\mu \xRightarrow{C} \nu$ iff $\mu = \mu_1 S \mu_2$ and $\nu = \mu_1 \alpha \mu_2$ and $S \rightarrow \alpha$. Let $\mu \xRightarrow{*}_C \nu$ denote that operation's reflexive, transitive closure. Then $L = \left\{ \omega \in \Sigma^* \mid V_0 \xRightarrow{*}_C \omega \right\}$ defines the context free language.

Example 1. A CFG (V, Σ, P, V_0) for arithmetic expressions can be defined by

- $V = \{E, T, F\}$
- $\Sigma = \{\oplus, \odot, (,), f\}$
- $V_0 = E$
- P defined as

$$E \rightarrow T \oplus E$$

$$E \rightarrow T$$

$$T \rightarrow F \odot T$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow f$$

An example for a valid sentence is $(f \oplus f) \odot f \oplus f$.

Definition 3. A String Template System (**STS**) can be defined as a tuple (T, Σ, R, T_0) where

- T is a finite set of string templates,
- Σ is a finite set of symbols, disjoint from T ,
- $R : T \rightarrow (\Sigma \cup T \cup \mathcal{P}(T))^*$ is a function with $\mathcal{P}(T) = \{U \mid U \subseteq T\}$ being the power set,
- $T_0 \in T$ is the expanded string template.

The symbol \mathcal{E} is used to denote a string template with an empty right-hand-side, i.e. $R(\mathcal{E}) = \varepsilon$. We denote the mapping $R(A) = \alpha$ as $A \mapsto \alpha$.

Definition 4. A String Template Language (**STL**) is the set of all strings that can be produced by a STS through recursive substitution of string templates with their mapping. For sets of string templates, any member may be expanded.

For $A \in T, \alpha \in \mathcal{P}(T), \beta, \mu, \nu \in (\Sigma \cup T \cup \mathcal{P}(T))^*$ we write

$$\begin{aligned} \mu \xRightarrow[S]{\Rightarrow} \nu &\Leftrightarrow (\mu = \mu_1 A \mu_2 \wedge \nu = \mu_1 \beta \mu_2 \wedge A \mapsto \beta) \\ &\vee (\mu = \mu_1 \alpha \mu_2 \wedge \nu = \mu_1 B \mu_2 \wedge B \in \alpha) \end{aligned}$$

We denote applications of the first alternative as $\mu \xRightarrow[S]{1} \nu$, of the second alternative $\mu \xRightarrow[S]{2} \nu$ and the reflexive, transitive closure as $\mu \xRightarrow[S]{*} \nu$. Then $L = \left\{ \omega \in \Sigma^* \mid T_0 \xRightarrow[S]{*} \omega \right\}$ denotes the string template language.

Example 2. A STS (T, Σ, R, T_0) for tuples can be defined by

- $T = \{S, F, C, E, D\}$
- $\Sigma = \{p, |, (,)\}$
- $T_0 = S$
- R defined as

$$\begin{aligned} S &\mapsto (F) \\ F &\mapsto E \{C, \mathcal{E}\} \\ C &\mapsto |F \\ E &\mapsto \{S, D\} \\ D &\mapsto p \end{aligned}$$

An example for a valid sentence is $(p|((p|p)|p))$.

Remark 1. The key difference between the structures for CFG and STS is found in the notation of alternatives. CFGs allow alternative definitions for meta symbols, but only a single definition on the right-hand-side of a rule, while STSs allow only a single definition for each string template but with multiple possible expansions.

```

1  def S() // S ↦ (F)
2    " («F» "
3
4  def F() // F ↦ E {C, E}
5    " «E» «IF c1» «C» «ENDIF» "
6
7  def C() // C ↦ |F
8    " | «F» "
9
10 def E() // E ↦ {S, D}
11    " «IF c2» «S» «ELSE» «D» «ENDIF» "
12
13 def D() // D ↦ p
14    " p "

```

Listing 3. *Xtend* string template for generating the tuple STS from example 2.

$$\begin{aligned}
 S &\mapsto \langle \text{body} \rangle \langle \text{h2} \rangle D_1 \langle \text{/h2} \rangle \{T_1, \mathcal{E}\} \langle \text{/body} \rangle \\
 T_1 &\mapsto \langle \text{h3} \rangle D_2 \langle \text{/h3} \rangle \langle \text{d1} \rangle \{T_2, \mathcal{E}\} \langle \text{/d1} \rangle \{T_1, \mathcal{E}\} \\
 T_2 &\mapsto \{T_3, T_4, \mathcal{E}\} \{T_2, \mathcal{E}\} \\
 T_3 &\mapsto \langle \text{dt} \rangle D_3 \langle \text{/dt} \rangle \langle \text{dd} \rangle = D_4 \langle \text{/dt} \rangle \\
 T_4 &\mapsto \langle \text{dt} \rangle \langle \text{dd} \rangle \langle \text{i} \rangle \text{Enumeration is extensible} \langle \text{/i} \rangle \langle \text{/dd} \rangle
 \end{aligned}$$

Listing 4. STS for listing 1 from the introduction. This rendering doesn’t account for whitespace and has no mappings defined for the dynamic expansions D_1 to D_4 .

2.2 Relation to Real-World String Templates

String templates typically feature embedded control structures for branching, looping and recursion as well as embedded variable references in expressions. During interpretation, the textual output is determined by evaluating the dynamic expressions and inserting their string representations into the output stream in sequence. Conditions attached to the control structures are evaluated as well and used to adjust the control flow accordingly. Examples for string template engines that function as described include *String Template* [7], rich strings in *Xtend* [2], and *Cheetah* for *Python* [8]. Both the evaluation of dynamic expressions and the selection of the applicable expansion are done referencing a meta-model instance.

Since the point of static analysis is to abstract from meta-model instances, Definition 3 captures the control structures with the power set for string templates and Definition 4 allows the control flow to pass to any of the included string templates within a power set. This arrangement allows the modeling of control structures as well as arbitrary other forms of selecting alternative expansions. Loops are a special case of recursion (tail-recursion) and can therefore also be expressed within the structure. The mathematical structure doesn’t capture dynamic expressions; we will disregard them until the relationship to context free grammars is understood more clearly.

Listings 3 and 4 provide examples for mapping between the specific *Xtend* notation for string templates and the structure from Definition 3. In the set of string template mappings in listing 4 all whitespace is omitted for brevity. The attribute references represented by D_1 to D_4 cannot be expressed as STS yet and therefore have an undefined mapping. We revisit dynamic expressions in Sect. 3.

2.3 Mappings

With the definitions in place, we can now ask the formal question: given a CFG $G = (V, \Sigma, P, V_0)$ describing the target language and a STS $S = (T, \Sigma, R, T_0)$ describing the code generator, can we decide if $L_S \subseteq L_G$?

Definition 5. *STS \mapsto CFG can be defined as follows: Define a function that takes the right-hand-side of a string template definition and expands it into a set of right-hand-sides with no alternatives. Use it for the definition of productions, since these allow multiple definitions but no alternative expansions.*

Formally, let $f : (\Sigma \cup T \cup \mathcal{P}(T))^ \rightarrow \mathcal{P}((\Sigma \cup T)^*)$ be*

$$f(\omega) = \begin{cases} \{\varepsilon\} & \text{for } \omega = \varepsilon \\ \{\omega\} & \text{for } \omega \in \Sigma \cup T \\ \{S \in \omega\} & \text{for } \omega \in \mathcal{P}(T) \\ \{\sigma b \mid b \in f(\beta)\} & \text{for } \omega = \sigma\beta, \sigma \in \Sigma \cup T, \beta \in (\Sigma \cup T \cup \mathcal{P}(T))^+ \\ \{A b \mid A \in \alpha, b \in f(\beta)\} & \text{for } \omega = \alpha\beta, \alpha \in \mathcal{P}(T), \beta \in (\Sigma \cup T \cup \mathcal{P}(T))^+ \end{cases}$$

Then $(V, \Sigma, P, V_0) = (T, \Sigma, \{S \rightarrow \alpha \mid S \in T, \alpha \in f(R(S))\}, T_0)$.

Lemma 1. *Every STS can be expressed as a CFG such that their respective languages are equal.*

Proof. Let L_S be the language of the STS $S = (T, \Sigma, R, T_0)$ and L_G the language of the CFG $G = (V, \Sigma, P, V_0)$ defined as outlined in Definition 5.

$L_S \subseteq L_G$ Let $T_0 \xRightarrow{*}_S \omega$, i.e. there is a sequence $T_0 \xRightarrow{1}_S \omega_1 \xRightarrow{2}_S \dots \xRightarrow{2}_S \omega_n = \omega$ with $\omega_1, \dots, \omega_n \in (\Sigma \cup T \cup \mathcal{P}(T))^*$. Without loss of generality, let the sequence be ordered such that for every application of the form $\mu_1 S \mu_2 = \omega_i \xRightarrow{1}_S \omega_{i+1} = \mu_1 \alpha \mu_2$, there is an immediate sequence of derivation steps $\omega_{i+1} \xRightarrow{2}_S \dots \xRightarrow{2}_S \omega_{i+j} = \mu_1 \alpha' \mu_2$ with $\alpha' \in (\Sigma \cup T)^*$. We can now rewrite the sequence as

$$\begin{array}{ccccccc} T_0 & \xRightarrow{1}_S \omega_1 & \xRightarrow{2}_S \dots \xRightarrow{2}_S & \omega_{j_2} & \xRightarrow{1}_S \dots \xRightarrow{2}_S & \omega_{j_3} & \xRightarrow{1}_S \dots \xRightarrow{2}_S & \omega_{j_{k-1}} & \xRightarrow{1}_S \omega_{j_k} & = \omega \\ \parallel & & & \parallel & & \parallel & & \parallel & & \\ \chi_1 & \xRightarrow{*}_S & & \chi_2 & \xRightarrow{*}_S & & \chi_3 & \xRightarrow{*}_S \dots & \chi_{k-1} & \xRightarrow{1}_S \chi_k \end{array}$$

Selecting an arbitrary step $\chi_i \xrightarrow[S]{*} \chi_{i+1}$, we reason:

$$\begin{aligned} \Rightarrow \chi_i &= \mu_1 S \mu_2 \wedge \chi_{i+1} = \mu_1 \alpha' \mu_2 \wedge S \xrightarrow[S]{1} \alpha \xrightarrow[S]{2} \dots \xrightarrow[S]{2} \alpha' \\ \Rightarrow \alpha' &\in f(R(S)) \\ \Rightarrow S &\rightarrow \alpha' \\ \Rightarrow \chi_i &\xrightarrow[C]{*} \chi_{i+1} \end{aligned}$$

We conclude that $T_0 \xrightarrow[C]{*} \omega$.

$L_S \supseteq L_G$ Let $T_0 \xrightarrow[C]{*} \omega$, i.e. there is a sequence $T_0 = \chi_1 \xrightarrow[C]{*} \dots \xrightarrow[C]{*} \chi_n = \omega$ with $\chi_1, \dots, \chi_n \in (\Sigma \cup T)^*$. Selecting an arbitrary step $\mu_1 S \mu_2 = \chi_i \xrightarrow[C]{*} \chi_{i+1} = \mu_1 \alpha \mu_2$, we conclude that $S \rightarrow \alpha$. Therefore $\alpha \in f(R(S))$ and $S \xrightarrow[S]{*} \alpha$. It follows that $\chi_i \xrightarrow[S]{*} \chi_{i+1}$. Since all steps have analogous derivations, we conclude $T_0 \xrightarrow[S]{*} \omega$.

Example 3. Given the STS (T, Σ, R, T_0) defined in Example 2 and following the construction in Definition 5, an equivalent CFG (V, Σ, P, V_0) is

- $V = T$
- $V_0 = T_0$
- P defined as

$$\begin{aligned} S &\rightarrow (F) \\ F &\rightarrow E \\ F &\rightarrow EC \\ C &\rightarrow |F \\ E &\rightarrow S \\ E &\rightarrow D \\ D &\rightarrow p \end{aligned}$$

Definition 6. $CFG \mapsto STS$ can be defined as follows: Define a function that takes a meta character from the left-hand-side of a production rule and defines the set of all possible expansions. Use it to add an indirection to the rules in R , using the mechanism of selecting an alternative during an expansion and adding new string templates as targets.

Formally, let $g : V \rightarrow \mathcal{P}((\Sigma \cup V)^*)$ be defined as $g(S) = \{\alpha \mid S \rightarrow \alpha\}$. Then (T, Σ, R, T_0) can be defined as

- $T = V \cup \{A_{S_i} \mid S \in V, i \in \{1, \dots, |g(S)|\}\}$
- $T_0 = V_0$
- $R = \{S \mapsto \{A_{S_i}\}, A_{S_i} \mapsto \alpha \mid i \in \{1, \dots, |g(S)|\}, \alpha \in g(S)\}$

Lemma 2. Every CFG can be expressed as a STS such that their respective languages are equal.

Proof. Let L_G be the language of the CFG (V, Σ, P, V_0) and L_S the language of the STS (T, Σ, R, T_0) defined according to Definition 6.

$L_G \subseteq L_S$ Let $T_0 \xRightarrow{*}_C \omega$, i.e. there is a sequence $T_0 \Rightarrow_C \omega_1 \Rightarrow_C \dots \Rightarrow_C \omega_n = \omega$ with $\omega_1, \dots, \omega_n \in (\Sigma \cup V)^*$. Selecting an arbitrary step $\mu_1 S \mu_2 = \omega_i \xRightarrow{*}_C \omega_{i+1} = \mu_1 \alpha \mu_2$, we conclude $S \rightarrow \alpha$. By construction, the derivation sequence $S \xRightarrow{1}_S \{A_{S_i}\} \xRightarrow{2}_S A_{S_j} \xRightarrow{1}_S \alpha$ is possible. Therefore $\omega_i \xRightarrow{*}_S \omega_{i+1}$ is valid as well. Since all steps have analogue derivation steps, we can conclude that $T_0 \xRightarrow{*}_S \omega$.

$L_G \supseteq L_S$ Let $T_0 \xRightarrow{*}_S \omega$, i.e. there is a sequence $T_0 \Rightarrow_S \omega_1 \Rightarrow_S \dots \Rightarrow_S \omega_n = \omega$ with $\omega_1, \dots, \omega_n \in (\Sigma \cup T \cup \mathcal{P}(T))^*$. By construction of R , we observe that the sequence for expanding any $S \in V$ must be $S \xRightarrow{1}_S \{A_{S_i}\} \xRightarrow{2}_S A_{S_j} \xRightarrow{1}_S \alpha$. This sub-sequence has an analogue in $S \Rightarrow_C \alpha$. Since $T_0 \in V$ and $\omega \in \Sigma^*$, i.e. all symbols have been expanded, we conclude $T_0 \xRightarrow{*}_C \omega$.

Example 4. Given the CFG (V, Σ, P, V_0) defined in Example 1 and following the construction in Definition 6, an equivalent STS is

- $T = V \cup \{A_{E_1}, A_{E_2}, A_{T_1}, A_{T_2}, A_{F_1}, A_{F_2}\}$
- $T_0 = V_0$
- R defined as

$$\begin{aligned}
 E &\mapsto \{A_{E_1}, A_{E_2}\} \\
 A_{E_1} &\mapsto T \oplus E \\
 A_{E_2} &\mapsto T \\
 T &\mapsto \{A_{T_1}, A_{T_2}\} \\
 A_{T_1} &\mapsto F \odot T \\
 A_{T_2} &\mapsto F \\
 F &\mapsto \{A_{F_1}, A_{F_2}\} \\
 A_{F_1} &\mapsto (E) \\
 A_{F_2} &\mapsto f
 \end{aligned}$$

Corollary 1. *Given Lemmas 1 and 2, STS and CFG are interchangeable notations for the same set of languages.*

Theorem 1. *Given an arbitrary STS S and an arbitrary CFG G , the problem $L_S \subseteq L_G$ is undecidable.*

Proof. The containment-problem is undecidable for context free languages [3]. Since the two formalisms have the same expressive power, this result applies.

Corollary 2. *Given an arbitrary CFG G , we can derive an equivalent STS S with $L_S = L_G$. Any subset $S' \subseteq S$ fulfills $L_{S'} \subseteq L_G$.*

3 Discussion

The equivalence in expressiveness between context free grammars and string templates established in Corollary 1 helps to explain the latter's popularity within the domain of code generation, since string templates are essentially context free grammars with imperative execution semantics and code generators are typically written in an imperative programming style.

Theorem 1 of the previous section also shows that even though arbitrary problem instances are undecidable, we can guarantee the syntactical correctness of the generated language by restricting the allowed set of string templates to subsets and language invariant transformations of the target language's associated STS (see Corollary 2). This provides a formal explanation for the results in Wachsmuth's paper [9] and their subsequent adaptation as part of Arnoldus' PhD thesis [1].

3.1 Capturing Dynamic Expressions

In order to capture dynamic expressions we can extend the structure in Definition 3 to support variable references with an associated mapping into strings. The interpretation from Definition 4 can then be adjusted to evaluate these newly allowed expressions through substitution with their image. This prompts the following changes to Definitions 3 and 4:

Definition 7. A String Template System with Expressions (**STSE**) can be defined as a tuple $(T, E, \Sigma, F, R, T_0)$ where

- T is a finite set of string templates,
- E is a finite set of expressions, disjoint from T ,
- Σ is a finite set of symbols, disjoint from T and E ,
- $F : E \rightarrow \Sigma^*$ is a function,
- $R : T \rightarrow (\Sigma \cup T \cup E \cup \mathcal{P}(T))^*$ is a function,
- $T_0 \in T$ is the expanded string template.

Definition 8. A String Template Language with Expressions (**STLE**) is the set of all strings that can be produced by a STSE through recursive substitution of string templates with their mapping and substitution of expressions with their mapping. For sets of string templates, any member may be expanded.

For $A \in T, \alpha \in \mathcal{P}(T), \beta, \mu, \nu \in (\Sigma \cup T \cup E \cup \mathcal{P}(T))^*$ we write

$$\begin{aligned} \mu \xRightarrow{S} \nu &\Leftrightarrow (\mu = \mu_1 A \mu_2 \wedge \nu = \mu_1 \beta \mu_2 \wedge R(A) = \beta) \\ &\vee (\mu = \mu_1 E \mu_2 \wedge \nu = \mu_1 \gamma \mu_2 \wedge F(E) = \gamma) \\ &\vee (\mu = \mu_1 \alpha \mu_2 \wedge \nu = \mu_1 B \mu_2 \wedge B \in \alpha) \end{aligned}$$

Then $L = \left\{ \omega \in \Sigma^* \mid T_0 \xRightarrow{S}^* \omega \right\}$ denotes the string template language with expressions.

Unfortunately, as Parr points out in [6], this makes the interpretation much more expressive, allowing for the generation of type 0 languages since F essentially introduces a Turing machine and is therefore impossible to verify statically [4]. In other words, allowing arbitrary strings in the image of F can invalidate any formal syntax.

However, restricting the image to regular expressions instead of arbitrary strings would continue to allow for static validation. In addition, unlike for context free grammars the containment problem is decidable for regular expressions, allowing us to assign arbitrary regular expressions to dynamic components without losing the ability to statically verify their correctness.

We would like to propose the addition of a dedicated regular expression type to the type system of domain-specific languages for code generation as a subclass of the generic string type in order to allow the static type-checker to verify the correctness of a dynamic expression with regards to the syntax of the target language in the context of a string template. Since grammars for real-world languages typically describe their terminals using regular expressions already, it would be feasible to support dynamic expressions at exactly these points in the associated STSE without sacrificing any static guarantees.

4 Conclusions and Outlook

We have formally defined sets of interconnected string templates and have shown their equivalence to context free grammars using a constructive proof that maps one onto the other. From this, we were able to conclude the undecidability of the general version of the problem to statically decide whether the language generated by a set of string templates is a subset of the target language. However, the construction has also allowed us to identify the conditions under which a static guarantee for syntactical correctness can be provided: if we prevent arbitrary sets of string templates to be used in the code generation and instead only allow those that can be proven to have an analogue derivation for the grammar of the target language. We have connected these results to previous literature and outlined a scheme to include support for dynamically evaluated expressions as well.

It remains to be seen if the proposed restrictions with regards to allowing typed dynamic expressions only at specific points in string templates will be acceptable to programmers or not. If not, an easy workaround for programmers would be a dynamic cast from an unrestricted string into a string with a compatible regular expression, restoring the test-based verification outlined by Arnoldus [1].

References

1. Arnoldus, B.J.: An illumination of the template enigma: software code generation with templates. Ph.D. thesis, Technische Universiteit Eindhoven (2010)
2. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing, Birmingham (2013)

3. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, Chap. 8, p. 203. In: [5] (1979)
4. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, Chap. 8, pp. 185–192. In: [5] (1979)
5. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)
6. Parr, T.J.: Enforcing strict model-view separation in template engines. In: Proceedings of the 13th International Conference on World Wide Web, pp. 224–233. ACM (2004)
7. Parr, T.J.: A functional language for generating structured text (2006). <http://www.cs.usfca.edu/~parrt/papers/ST.pdf>
8. Rudd, T., Orr, M., Bicking, I., Esterbrook, C.: Cheetah: the python-powered template engine. In: 10th International Python Conference-2002 (2007)
9. Wachsmuth, G.: A formal way from text to code templates. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 109–123. Springer, Heidelberg (2009). doi:[10.1007/978-3-642-00593-0_8](https://doi.org/10.1007/978-3-642-00593-0_8)

SDL 2017: Model-Driven Engineering for Future Internet
18th International SDL Forum, Budapest, Hungary,
October 9–11, 2017, Proceedings
Csöndes, T.; Kovács, G.; Réthy, G. (Eds.)
2017, XI, 173 p. 74 illus., Softcover
ISBN: 978-3-319-68014-9