Static Syntax Validation for Code Generation with String Templates

Paper Contribution to SDL 2017

Dorian Weber   Joachim Fischer

Institut für Informatik
Humboldt-Universität zu Berlin

October 9th, 2017

Observations

- core tenet of model-driven engineering is the use of domain models to represent abstract knowledge about a particular application domain
- applications ultimately require a model-to-model or model-to-text transformation

Problem

- performing a model-to-text transformation: what can we say about the generated text?
  1. does its syntax conform to some context-free grammar?
  2. are identifiers declared before being used?
  3. do their data types permit the use in generated operations?
  4. can the execution result in undefined behaviour?
- questions like these may only be answered with an automated proof, not through testing
- we focus on syntax

HTML in Xtend

```
'''
  <body>
    <h2>«module.name»</h2>
    «FOR expr: module.member»
      <h3>«expr.name»</h3>
      <dl>
        «FOR it: expr.member»
          «IF it.univerval»
            <dt>«it.name»</dt>
            <dd>=«it.value»</dt>
          «ELSEIF it.extensible»
            <dt />
            <dd><i>Enumeration is extensible</i></dd>
          «ENDIF»
        «ENDFOR»
      </dl>
    «ENDFOR»
  </body>
'''
```

C enumeration in Xtend

```
'''
    typedef enum «expr.name» {
        «FOR it: expr.member»
            «IF it.univerval»
                «it.name» = «it.value»,
            «ELSEIF it.extensible»
                /*
                 * Enumeration is extensible
                 */
            «ENDIF»
        «ENDFOR»
    } e_«expr.name»;
'''
```

Research Question (informal)

Prior to seeing any meta-model instance, under which circumstances can we guarantee that a string template will expand to syntactically correct code?

Context Free Grammar

Definition (CFG)

A *Context Free Grammar* 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 $S \to \alpha$.

Context Free Language

Definition (CFL)

A *Context Free Language* is the set of all strings that can be produced by a CFG through the application of a sequence of production rules to the start symbol via substitution of a meta character by the rule's right-hand-side.

Example (Arithmetic Expression CFG)

$G_{\text{Expr}} = (V, \Sigma, P, V_0)$ where $V_0 = E$ and P defined as

$$
\begin{aligned}
E &\rightarrow T \oplus E \\
E &\rightarrow T \\
T &\rightarrow F \odot T \\
T &\rightarrow F \\
F &\rightarrow \langle E \rangle \\
F &\rightarrow t
\end{aligned}
$$

- $\langle t \oplus t \rangle \odot t \oplus t \in L_{\text{Expr}}$

String Template System

Definition (STS)

A *String Template System* 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,
- $T_0 \in T$ is a 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 $A \mapsto \alpha$.

String Template Language

Definition (STL)

A *String Template Language* is the set of all strings that can be produced by a STS through recursive substitution of string templates with their mapping, beginning at the start template. For sets of string templates, any member may be expanded.

Example (Tuple STS)

$S_{\text{Tup}} = (T, \Sigma, R, T_0)$ where $T_0 = S$ and R defined as

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

- $\langle p|\langle\langle p|p\rangle|p\rangle\rangle \in L_{\text{Tup}}$

STS → Xtend

```
def S() // S ↦ (F)
        '''(«F»)'''

def F() // F ↦ E {C, ℰ}
        '''«E»«IF c1»«C»«ENDIF»'''

def C() // C ↦ |F
        '''|«F»'''

def E() // E ↦ {S, D}
        '''«IF c2»«S»«ELSE»«D»«ENDIF»'''

def D() // D ↦ p
        '''p'''
```

Xtend → STS

$$S \mapsto \texttt{<body><h2>}D_1\texttt{</h2>}\,\{T_1,\mathcal{E}\}\,\texttt{</body>}$$

$$T_1 \mapsto \texttt{<h3>}D_2\texttt{</h3><dl>}\,\{T_2,\mathcal{E}\}\,\texttt{</dl>}\,\{T_1,\mathcal{E}\}$$

$$T_2 \mapsto \{T_3,T_4,\mathcal{E}\}\,\{T_2,\mathcal{E}\}$$

$$T_3 \mapsto \texttt{<dt>}D_3\texttt{</dt><dd>=}D_4\texttt{</dt>}$$

$$T_4 \mapsto \texttt{<dt/><dd><i>Enumeration is extensible</i></dd>}$$

Research Question (formal)

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* $L_S \subseteq L_G$*?*

Lemmas

Lemma (STS $\subseteq$ CFG)

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

Lemma (STS $\supseteq$ CFG)

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

Corollary (STS $=$ CFG)

STS and CFG are interchangeable notations for the same set of languages.

Mapping from STS to CFG

Example (Tuple CFG)

$(T, \Sigma, R, T_0) = G_{\mathrm{Tup}} \mapsto S_{\mathrm{Tup}} = (V, \Sigma, P, V_0)$ with P defined as

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

Mapping from CFG to STS

Example (Arithmetic Expression STS)

$(V, \Sigma, P, V_0) = S_{\text{Expr}} \mapsto G_{\text{Expr}} = (T, \Sigma, R, T_0)$ with R defined as

$$
\begin{array}{lcr}
 & & E \mapsto \{A_{E_1}, A_{E_2}\} \\
E \to T \oplus E & & A_{E_1} \mapsto T \oplus E \\
E \to T & & A_{E_2} \mapsto T \\
 & & T \mapsto \{A_{T_1}, A_{T_2}\} \\
T \to F \odot T & \longrightarrow & A_{T_1} \mapsto F \odot T \\
T \to F & & A_{T_2} \mapsto F \\
 & & F \mapsto \{A_{F_1}, A_{F_2}\} \\
F \to \langle E \rangle & & A_{F_1} \mapsto \langle E \rangle \\
F \to t & & A_{F_2} \mapsto t
\end{array}
$$

Conclusion

Theorem

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

Corollary

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

Trojan Horse

- string templates support control structures and string literals, but no attribute references
- include references to meta-model instances into the formal structure for string templates

String Template System with Expressions

Definition (STSE)

A *String Template System with Expressions* 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,
- $F : E \to \Sigma^*$ is a function,
- $R : T \to (\Sigma \cup T \cup E \cup \mathcal{P}(T))^*$ is a function,
- $T_0 \in T$ is the expanded string template.

String Template Language with Expressions

Definition (STLE)

A *String Template Language with Expressions* 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.

Discussion

- instead of permitting Type-0 languages in dynamic expressions, we have to restrict them to Type-2 or higher
- two potential avenues:
  1. test dynamic expressions before substituting
  2. prove in advance that the expression conforms to the target syntax

Outlook

```
grammar HTML {
    ROOT -> "<html>" BODY "</html>";
    BODY -> "<body>" (HEAD | ...) "</body>";
    HEAD -> "<h1>" [^<]* "</h1>" | ...;
    ...
}

HTML doc = '''
    <html>
        <body>«heading("SDL 2017")»</body>
        ...
    </html>
''';

def heading(String<[^<]*> name)
    '''<h1>«name»</h1>''';
```