Towards Automatic Generation of a Coherent TTCN-3 Template Framework

Roland Gecse

Ericsson Hungary Ltd., P.O.Box 107, H-1300 Budapest 3, Hungary
roland.gecse@ericsson.com

Abstract. The template framework is a subset of TTCN-3 template declarations, which determines the entire template hierarchy within an abstract test suite. This paper introduces a deterministic method for generating such a template framework from type definitions.

1 Introduction

Test data and particularly TTCN-3 [1] templates account for significant part of test suite design both in terms of size and development time. The template framework expresses the relationship between the individual templates. It consists of template declarations, which describe the messages required for implementing dynamic test behaviour. TTCN-3 offers dedicated language constructions for organizing the template framework such as template parameterization or modification. A well-designed template framework is a precondition to concise templates, which are compact, well-organized and easy to maintain.

The template framework should be designed before test suite development with several requirements in mind. Excessively short or long as well as similar template declarations should be avoided. The template parameterization and modification shall also be used moderately. The perfect arrangement is hard to find because the requirements are often contradictory. It is thus desirable to have an automated template framework generation method even when the obtained result is not optimal.

The present article intends to facilitate test design by introducing the means and methods for generating an initial template framework from given type definitions with the ambition of assisting testers especially in the early phase of test suite design.

2 Type Structure Graph (TSG)

The TSG represents the encapsulation of structured types in order to reason about the hierarchy of the generated template framework. It captures the essential structural information but neglects some otherwise important properties such as the number and ordering of elements within a type. The simple and enumeration types are not container types so these are excluded from the model. The sub-type constraints of input type definitions are also ignored in TSG.

M. Núñez et al. (Eds.): TESTCOM/FATES 2009, LNCS 5826, pp. 223–228, 2009.

Formally, the $TSG = (V, E)$ is a directed graph whose V vertices model structured type definitions and $E \subseteq V^2$ directed edges express containment. An $uv \triangleq (u, v) \in E$ edge means that u includes v. The vertices of TSG are subdivided depending on the modeled structured type into pairwise disjoint sets $V_R = \{\texttt{record} \vee \texttt{set}\ types\}$, $V_C = \{\texttt{union}\ types\}$ and $V_L = \{\texttt{record of} \vee \texttt{set of}\ types\}$, such that $V_R \cup V_C \cup V_L = V$. The sets V_R, V_C and V_L are called record, choice and list nodes, respectively. The vertices of the graph are labeled with the structured type identifiers. The TSG edges are categorized, similarly to vertices, into pairwise disjoint sets $E_R = \{uv : u \in V_R\}$, $E_C = \{uv : u \in V_C\}$ and $E_L = \{uv : u \in V_L\}$, such that $E_R \cup E_C \cup E_L = E$. The sets E_R, E_C and E_L are named record, choice and list edges, in the order of definition.

The TSG may include self-loops and cycles as TTCN-3 supports recursive types. Multiple edges can also appear because a structured type definition may contain many identical type elements. The TSG is disconnected when the modeled type hierarchy includes independent messages. The sinks of TSG represent those structured types, which consist of simple type elements only. The sources of TSG do not necessarily correspond to the Protocol Data Units (PDUs). Hence, the $V_D \subseteq V$ set is used to mark those distinguished types for which the templates are declared. The PDU types, which represent the initial content of V_D, are obtained from the TTCN-3 port type definitions.

Certain TSG properties reveal interesting features of type hierarchy. The graph diameter, for instance, corresponds to the maximal depth of type hierarchy, which is a measure of encapsulation. The out-degree and the all-pairs shortest path length distribution of nodes express the size and complexity of type hierarchy.

The presence of Strongly Connected Components (SCCs) in TSG indicates recursion. The introduced method, however, does not intend to generate templates for recursive structures. However, the vertices of SCCs are saved into the set V_S as these require special attention during template framework generation.

3 Preliminaries of Template Generation

The template framework is constructed using an incremental method without backtracking. The **join** procedure merges templates in order to reduce the small ones while the **makepar** adds formal parameters to templates to avoid the large ones. The algorithm in Section 4 performs a careful analysis of arguments to determine the procedure to be invoked during the generation.

3.1 Template Join

The **join**$(t(u), t(v))$ procedure creates some new templates for type u by attaching the $t(v)$ templates to the $t(u)$ templates at each reference to v. The resulting templates replace the original content of $t(u)$ but $t(v)$ is not modified. The formal parameter lists of argument templates are also merged when necessary. The templates of u and v can be joined if TSG has at least one $u \cdots v$ path. If there

exists only an uv edge then $t(v)$ is joined to $t(u)$ directly at u. Otherwise, the $t(u)$ and $t(v)$ templates are joined recursively, in reverse topological order of edges along the set of all possible $u \cdots v$ paths.

The exact method of joining the templates of some neighbouring u, v nodes along the uv edge depends on the type of u, the $|t(u)|, |t(v)|$ number of templates for the involved nodes and the $|uv|$ number of connecting edges. If $u \in V_R$ then the $|t(v)|$ pieces of templates are attached to all $|uv|$ places in every $|t(u)|$ templates. Since each template can appear in many positions at the same time, the number of resulting templates can increase drastically. If $u \in V_C$ then u gets a new template declared for each $t(v)$ attached to all $|uv|$ places. The number of obtained templates is less than in case of records because the choice templates consist of a single field only. The $u \in V_L$ nodes are never joined with their elements. The number of templates after joining along the uv edge can be calculated with Equation 1.

$$|t(u)| \leftarrow \begin{cases} |t(u)| \cdot |t(v)|^{|uv|} & \text{if } u \in V_R \\ |t(u)| + |t(v)| \cdot |uv| & \text{if } u \in V_C \end{cases} \tag{1}$$

3.2 Template Parameterization

The **makepar**$(t(u), v)$ procedure creates formal parameters from all type v fields of $t(u)$ templates. The newly created parameters are appended to the existing formal parameter list of templates. The number of templates in $t(u)$ does not change during the execution. Similarly to the join procedure, performing the parameterization is only possible if v is reachable from u. The parameter references are inserted at those u' nodes for which $u \cdots u'v$ path exists. If there are many $u \cdots v$ paths then more type v parameters are appended. The original content of $t(u)$ is overwritten at u' elements by a reference to the new type v parameter. If v becomes a formal parameter then it must also be added to V_D so that its $t(v)$ templates are preserved during the generation.

4 Algorithm

The introduced direct method produces a template framework from the type definitions. The created templates together cover the entire TSG except the SCCs. The coverage of a template is defined as the set of TSG nodes, whose fields appear in the template declaration. The resulting templates may partially overlap and can also be parameterized but template modification is avoided. The algorithm comprises of three steps. Initially, a set of default templates is declared for each node. Next, the TSG is partitioned into subgraphs along the choice and list edges. The obtained subgraphs are rather small and include all record edges of TSG. The default templates of subgraph nodes are merged into subgraph templates using the **join** and **makepar** procedures. The subgraph templates cover all nodes and record edges of TSG. The final step assembles the template framework by joining the subgraph templates along the remaining choice and list edges either statically or dynamically using parameterization.

4.1 Default Template Generation

The default templates are declared for the $V \setminus V_S$ nodes of TSG such that the nested assignment of structured type fields is avoided.

The present approach assigns predefined values to all simple type fields, which remain unmodified during the template generation. Hence, the value used at initialization represents the final value of that field in all templates of the framework. The content of structured type fields of default templates can be set arbitrarily (e.g. to any value matching) as these get overwritten during generation.

The number of default templates varies by type. The records get only a single default template declared. The choices, however, get a separate default template generated for each field. The list nodes get a single parameterized template assigned, in which the formal parameter list comprises of the list type itself. The obtained default templates serve as input for the subgraph template generation.

4.2 Subgraph Template Generation

The generation of the template framework begins with splitting the TSG into TSG$[S_i]$ vertex induced subgraphs. The S_i nodes correspond to the (S_i, E_i) subgraphs obtained by excluding the $E_C \cup E_L$ edges from TSG. The induced subgraphs have pairwise disjoint vertex sets such that $\forall i, j : S_i \cap S_j = \emptyset$ and $\bigcup_i S_i = V$. The adjacent subgraphs are connected with the $E_B = E \setminus (\bigcup_i E_i)$ edges whose terminating vertices are called entry nodes. The SCCs are always considered independent subgraphs regardless of their internal structure.

The goal of subgraph template generation is to declare a minimal number of templates covering the maximal number of subgraph nodes. The default templates of adjacent nodes are joined recursively in bottom up order along the E_i edges until all entry nodes have their subgraph templates ready (Figure 1). According to Equation 1, the **join** procedure may create a huge number of templates even when the parameters (i.e. $|t(u)|, |t(v)|$ and $|uv|$) are small. Therefore, **join** is performed only if the join condition (Equation 2) is satisfied.

$$\begin{cases} |t(u)| = |t(v)| = 1 \vee |t(v)| = |uv| = 1 \vee |t(u)| = |uv| = 1 & \text{if } u \in V_R \\ |t(v)| = 1 \vee |uv| = 1 & \text{if } u \in V_C \end{cases} \tag{2}$$

The join condition constrains the template generation to permit only an additive increase in the number of resulting templates. It allows at most one of the three parameters to exceed one for V_R nodes. The join condition for V_C nodes is less restrictive as it does not include $t(u)$. When Equation 2 is not satisfied then the **makepar** procedure is invoked instead of **join**. It adds type v fields to formal parameters of all templates in $t(u)$ and promotes them to the V_D distinguished types.

The default templates, which could not be joined with any other templates become subgraph templates. The original default templates in $t(v)$ can be discarded after processing the templates of all predecessors if $v \notin V_D$.

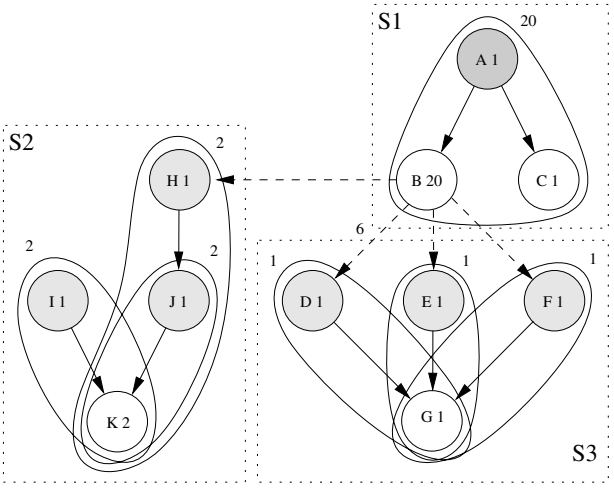

Fig. 1. An extract from an imaginary TSG: All entry nodes are shaded; vertex A is a distinguished node. The number of default templates appears next to the name inside the nodes. The S1, S2, S3 subgraphs and the generated subgraph templates are marked distinctively. S1 is covered by 20 overlapping subgraph templates while S2 has 2, and S3 contains 1 subgraph template for each entry node. The template framework generation algorithm joins the $t(H), t(D), t(E), t(F)$ templates with $t(B)$ and produces 21 templates for A (not shown in the Figure). The $2 + 2$ subgraph templates of I, J nodes are also preserved.

4.3 Template Framework Generation

The template framework is assembled from the templates generated for the TSG$[S_i]$ subgraphs (Figure 1). The S_p, S_c adjacent subgraphs are processed in reverse topological order such that S_p must not be a subgraph of an SCC (S_c can be any subgraph of TSG). The algorithm iterates through the entry nodes of the S_c subgraph and processes the $xy \in E_B : x \in S_p, y \in S_c$ edges one by one. It locates those $T \subseteq t(S_p)$ subgraph templates, which cover the xy edge (i.e. include a reference to node x containing y). The types of T templates are collected into set $W \subseteq S_p$. Clearly, W consists of those nodes of S_p subgraph, which have a path including the xy edge. The node x may not even appear in W because its default templates may have been joined with some of its predecessors' templates or became parameters during the subgraph template generation. If $y \in V_S$ then **makepar**(T, y) is invoked to make all references to the SCC node formal parameters in all referencing templates. Otherwise, the algorithm examines the $w \in W$ types of T templates.

If the processed w type is a record then all of its $t \in t(w) \cap T$ templates are joined with the $t(y)$ templates of the given entry node provided the join condition is satisfied. The join condition for records in Equation 2 normally checks $|t(x)|, |t(y)|$ and $|xy|$ properties of the edge. This time, it can happen

that $w \neq x$ meaning that the $t(w) \cap T$ and $t(y)$ arguments of the join are not adjacent templates. If the w and y nodes are not adjacent then the join needs to be performed along all the $w \cdots xy$ paths as described in Section 3. Consequently, the join condition for records needs to be interpreted differently. The selected templates of w are processed one by one, thus $|t| = 1$ needs to appear in the join condition instead of $|t(x)|$. Furthermore, the $|xy|$ has to be replaced with the number of references to xy in t, which is denoted by N. The join condition depends thus on N and $|t(y)|$ only. Therefore, the $\mathbf{join}(\{t\}, t(y))$ is only performed if $N = 1 \vee |t(y)| = 1$. If the join condition does not hold then $\mathbf{makepar}(\{t\}, y)$ is executed in order to replace all references to type y with parameters in template t.

If the processed w type is a choice then the $\mathbf{join}(t(w) \cap T, t(y))$ is executed once the join condition in Equation 2 is satisfied. When joining the subgraph templates is not feasible then the references to y become formal parameters by invoking $\mathbf{makepar}(t(w) \cap T, y)$.

Finally, at the end of generation all unnecessary templates, which are not declared for distinguished types, are removed.

5 Conclusion

The introduced deterministic algorithm creates some subgraph templates covering the adjacent record nodes of TSG and constructs the template framework by joining or parameterizing the obtained templates. The join is performed only if the number of resulting templates increases moderately. The parameterization is used to rule out a multiplicative increase of template declarations. The final values of simple type fields are set to dummy values by default because the protocol type definitions alone are insufficient for assigning template elements with semantically correct data. A more advanced method of value assignment exists but its discussion exceeds the limits of this article.

The template framework generation algorithm has been implemented as a prototype plugin of TITAN TTCN-3 Test Executor. It has been evaluated on some protocols with satisfactory results, which proved that it is possible to create a proper template framework exclusively from the type hierarchy. Nevertheless, the framework generation can be further optimized using the methods and infrastructure presented in [2]. The algorithm could be improved to take the statistical properties of abstract protocol syntaxes into account. The generated framework could be refined interactively during test suite development. A comparison with heuristic algorithms would also be meaningful.

References

1. Methods for Testing and Specification; The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language, ETSI ES 201 873-1 Edition 3.4.1 (2008)
2. Wu-Hen-Chang, A., Le Viet, D., Bátori, G., Gecse, R.: High-level restructuring of TTCN-3 test data. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 180–194. Springer, Heidelberg (2005)