Data Flow Testing in TTCN-3 with a Relational Database Schema

Gusztáv Adamis[1], Antal Wu-Hen-Chang[1], Gábor Árpád Németh[1], Levente Erős[2], and Gábor Kovács[2]

[1] Ericsson Hungary
Irinyi J. u. 4-20, H-1117, Budapest, Hungary
{gusztav.adamis,antal.wu-hen-chang,gabor.arpad.nemeth}@ericsson.com
[2] Department of Telecommunications and Media Informatics
Budapest University of Technology and Economics
Magyar tudósok körútja 2, H-1117, Budapest, Hungary
{eros,kovacsg}@tmit.bme.hu

Abstract. In script based testing traditionally test data and test environment parameters are stored either in the test script document or in a separate file. In some test execution systems, test data items are loaded to a database during the initialization, however that set of data remains static during the execution. In this paper, we propose a TTCN-3 based approach that stores all test case related data, even constants, local variables and parameterized message templates of a test case in a relational database. Data types and data type instances are all mapped into SQL schemas. When executing a test case, appropriate test templates are fetched or even generated on-the-fly for the subsequent test step, which results in a higher data flow coverage. The course of test events are logged in the database that makes reproduction possible.

Keywords: TTCN-3, test data, data flow, relational database.

1 Introduction

Web portal development frameworks such as Django [1], Ruby on Rails [2] and many others have an incorporated script based test environment. As web portals usually have a database back-end as data source in their production environment, test data to be used in the scripts defined in plain text files must be loaded into the test database before the test execution. Test data definition files contain finite sets of records of the data types used within the portal, and it is possible to execute the same test case with different test data records. Once loaded, these records can be accessed, fetched and modified during the execution of a test case, however the changes made to them cannot be traced back, as they are restored to their initial values upon the initialization of the next test case. Other limitations of these frameworks are that: the test data items must be given manually and therefore have fixed initial values; and there is no obvious way to assign a random value to a field of a record, which is transparent for the current test case.

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 1–18, 2013.

The UML and the UML Testing Profile are proposed in [3] for the data-driven testing of object-oriented models. These together provide the means for defining values in the test specification resulting in reusable and maintainable tests. Several testing frameworks exist already with the ability to separate data. SoapUI [4] is a general purpose testing framework that allows the user to use data sources with many formats, such as XML, Excel configuration files or databases. During test execution, test data items are read from the data source independently from the test steps defined in a separate document. MBUnit [5] is a .NET unit testing framework that also supports the separation of the data flow. Beyond being able to use hard-coded test data values, it is capable of getting test data from external sources such as XML spreadsheets or CSV files. EasyTest [6] is another new open source testing framework that allows for writing data-driven tests in JUnit. The Fitnesse framework [7] is primarily for testing requirements, and it already supports data-driven testing.

In TTCN-3, test data is not separated from the control flow of test cases as much as in the case of web development frameworks. Test data items appear within the TTCN-3 test cases themselves. Implementation and environment parameters are loaded from configuration files, record type and template definitions are vital parts of the TTCN-3 module, and the values used by local variables and constants are hardcoded into the test case part of the module.

In this paper, we introduce a framework for testing telecommunication protocols on the pattern of web development test environments, we put all the test data into a relational database that we access from test case bodies during the test execution. The rationale behind this approach is that many higher layer protocols are stateless or have only a very few control states, but at the same time they use complex protocol data units. When conformance testing such protocols, the very same message sequences are used over and over again, but with different test data. This paper investigates how the test data can be stored in a database, and if values for complex data structures can be generated on-the-fly based on user defined records of that database. The record type definitions available for the TTCN-3 test case are mapped to an SQL metamodel, and parameterized templates are records in those tables. We consider that all interactions of the test case take place by means of using parameterized templates, and fetch the parameters before the send and receive operations with an external function. Parameterized templates allow a higher level of flexibility compared to templates with hardcoded values, because parameters not yet accessed can be chosen at random at each execution providing a better data flow coverage than the single hardcoded value.

The cost of this flexibility with the parameterized templates is that we must provide a way to trace back the template value used in a test step and loose the portability of test cases. Therefore to enable persistent logging, we define the formal graph model of a test case including alt statements, branches, loops and local variables and constants that we map to an SQL metamodel as well. This allows the on-the-fly tracking of changes of values of local variables, so that in subsequent test steps the proper value can be used in templates.

The paper is organized as follows. Section 2 defines the architecture of the framework presented. In Sect. 3, we show the graph model of the test case we use in the database to represent the control flow of test cases. Section 4 gives the SQL metamodel for storing record types and parameterized templates, and describes the SQL metamodel we store this test case representation in. How a test case is transformed and executed within this framework is shown in Sect. 5, where we show an example where our method can be useful: a flow control counter overflow check. Finally, Sect. 6 summarizes the paper.

2 A Framework for Storing TTCN-3 Test Data in a Relational Database

The fundamental idea behind this framework is to completely separate test data from the test cases, so that each test case rewritten according to the concepts presented in this paper can be executed with arbitrary number of possibly random test data items without a change. Therefore, we remove all occurrences of values, variable and template accesses from original test case and replace them with external TTCN-3 function calls that implement the fetching and storing of these data items in a database.

When storing test data separately, a database has several benefits over using a simple configuration file. Meta-information such as the type of a field or of a variable can be encoded in the database schema. The complex data structures, or even protocol data units (PDUs), are constructed based on the user defined data, are used in an interaction. These structures or PDUs can be stored in the database and reused when repeating the test. In a long sequence of interactions, the user defined initial values may change, the database helps with tracking these changes, thus the database can store snapshots of the variable vector and associate that with a state of the interaction. Another advantage is that the database manager understands the natural ordering of primitive types such as integer, and for instance boundary values with regard to a table constraint can be very easily fetched.

Our method consists of two phases. First, the input TTCN-3 documents are parsed, and graph models are extracted from them: one that represents the data types available for the test cases, and for each test case a graph that represent the control flow. These pieces of information are then loaded into the database. The second phase is the test execution itself, where the evaluation of expressions and the parameterization of templates are performed within the database.

Figure 1 shows the process of the initialization phase. The input of the process is a set of TTCN-3 documents. These documents are parsed with a customized TTCN-3 parser, and a test data model and test case models are extracted, which are later on loaded into the metadata tables of the database. If data constraints have been defined on the datatypes and the database server supports constraint specification, the schema is updated. We use a custom mapping for that, not a nowadays standard way of object relational mapping (ORM). Transformations are performed on the input TTCN-3 documents that preserve the test case behavior, for each data type definition a set of parameterized template definitions

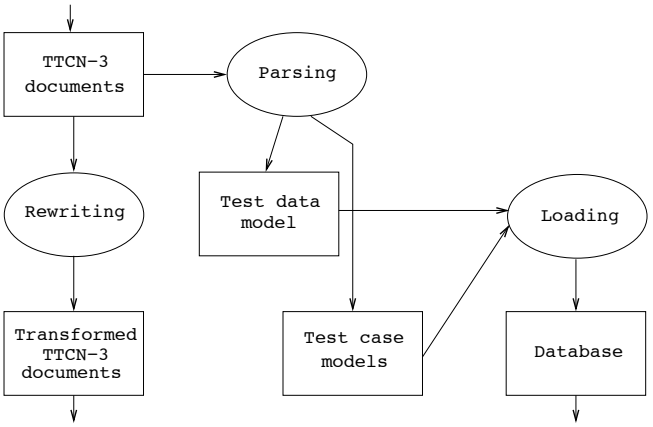

Fig. 1. The initialization of test execution

are generated that are used in `send` and `receive` operations. All variable accesses are replaced with calls to external functions. The outputs of this phase are the transformed TTCN-3 documents and the information stored about the test data and test cases in the database.

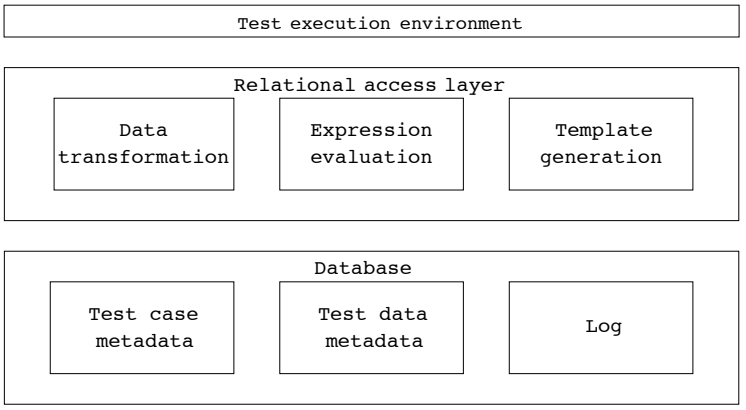

Fig. 2. The layered architecture of the framework, and the main function groups used in test execution

Figure 2 shows the architecture of our framework. When executing a test case, the test execution environment [8] accesses the database by calling the functions of the relational access layer implemented with external TTCN-3 functions. The relational access layer has three main function groups. In the transformed test case, a value can be accessed in three different ways. Data transformation that implements the relational mapping transparently maps database records to native TTCN-3 values and vice versa. When a value is given as an expression with

several variable identifiers, the expression evaluation module resolves the variables, transforms the operator in the expression to SQL syntax, and retrieves the value by executing an SQL query built on the pattern of the expression. The third method of value generation is the use of parameterized templates, which are built on-the-fly with Gecse's [9] method. If a field of a template has not yet been defined in previous interactions of the test case, a value is generated possibly at random and stored in the database for future re-use. The database layer is responsible for storing the data type, template and test case models, and logs the current and previous snapshots of test states of the execution. This latter logging allows the identification of freely definable variables and template fields. The database layer is completely hidden from the test engineers, who write the TTCN-3 test cases. They have to follow a well-defined set of conventions, but are not required to have specific database or SQL knowledge.

In the following, we introduce the data type and test case metamodels we create and map to the database after the parsing of the original TTCN-3 documents.

3 TTCN-3 Data and Test Case Representation

In this section, we define two graphs we use to represent TTCN-3 type and template definitions and TTCN-3 test case definitions. These models can automatically be generated from a TTCN-3 document with a parser and a code generator.

3.1 TTCN-3 Data Representation

In this paper, we use the concept of the type structure graph defined by Gecse's [9]. We model data types with a forest of directed graphs, however our focus is limited to TTCN-3 primitive and record types and TTCN-3 `record of`, `set of`, `set` and `union` types are left out of consideration. We denote the collection of data types with $D = (N, E, L)$ such that $E : N \times L \times N$, where a node $n \in N$ represents a data type, and an outgoing edge $e = (n_i, l, n_j)$, $e \in E$, $n_i, n_j \in N$, $l \in \mathbb{N}$ denotes that the data type n_i has field of the type represented by the target node n_j, and the $l \in L$ label defines the field name and an ordering constraint on the edges. Note that D is not necessarily a tree, the data type definitions may include recursive associations that result in strongly connected cliques.

The set of walks returned by the function $\mathsf{walk} : N \to W$ called subgraph templates in Gecse's work is an unbounded set of finite trees generated by traversing recursively all outgoing edges at each visited node starting from a node $n \in N$ until any node $n_i \in N$, $N_{primitive} \subseteq N$ representing a primitive type is reached. A walk $w \in W$, $W = \mathsf{walk}(n)$, $n \in N$ corresponds to a TTCN-3 parameterized template of type N. A data type instance or a template for walk $w \in W$ is generated by recursively applying the function $\mathsf{val} : W \to \mathcal{D}$ to all subtrees of W, where $\mathcal{D}$ is the domain of W rooted in type N. The function $\mathsf{type} : W \to N$ returns the type of a template, and the function $\mathsf{type} : V \to N$ gives the type of a variable.

```
1  type record M {
2   A a,
3   C c,
4   octetstring d
5  }
6
7  type record A{
8   octetstring a
9  }
10
11 type record C{
12  integer t,
13  integer s,
14  integer r
15 }
16
17 template C t_c (integer v_t
       , integer v_s, integer
       v_r) {
18   t := v_t,
19   s := v_s,
20   r := v_r
21 }
```

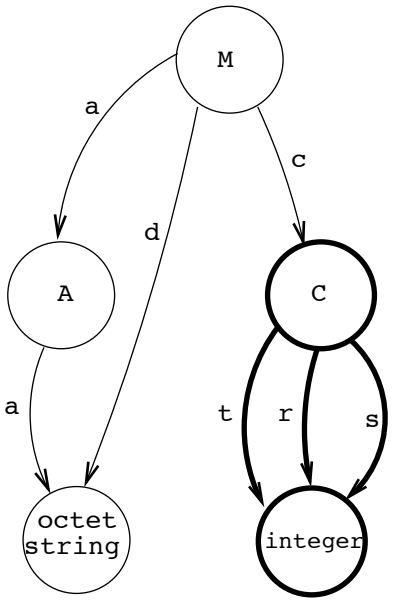

Fig. 3. Graph model of the data types. On the left a TTCN-3 code snippet with some type definitions can be seen, on the right the relationship graph between those types and a sub-template graph with bold lines are shown

Figure 3 shows a sample TTCN-3 type definition code snippet on the left and its graph representation after parsing on the right. In this example, five data types are used, three of them are defined here and two of them are primitive types, so $N = \{M, A, C, \mathsf{integer}, \mathsf{octetstring}\}$. The edge $e = (A, a, \mathsf{octetstring}) \in E$ for instance shows that type A has a field named a of type octetstring. Bold lines represent a walk $t_c \in W$, which is the only available walk from node C, this corresponds to the parameterized template named t_c. This t_c has a subgraph template rooted in node C. Note that the graph model is not necessarily acyclic.

3.2 TTCN-3 Test Case Representation

Mathematically, TTCN-3 test cases can be unfolded into a tree structure that is not necessarily finite. A common model in the literature for representing such trees is a rooted, edge labelled, directed graph, or more specifically a test transition system as defined in [10].

We, in this paper, use a different approach to represent the control flow of test cases. Our model of a test case $TC = (S, P, V, I, O, T)$ is a rooted edge and node labelled directed graph with two types of nodes, which is closely related

to the EFSM model used in [11]. The not necessarily finite set S represents the set of end nodes (test states) of a test step. The other type of node set is the set P that represents the set of branching nodes between test step nodes that correspond to TTCN-3 `if..else if..else` and `select..case` constructs. The set T represents the set of edges of the graph. The sets V, I and O represent the sets of variable identifiers, input template names and output template names respectively. Each $v \in V$ is associated with a type $n \in N$ and a value $\mathsf{val}(\mathsf{walk}(n))$.

Labeling functions assign labels to the nodes and edges of the graph. Each $s_l \in S$ node with the out-degree $deg_{out}(s_l) = 0$ is labelled with the function $\mathcal{L}_s : S \rightarrow \{\mathsf{pass}, \mathsf{fail}, \mathsf{inconc}\}$, which corresponds to the TTCN-3 `setverdict` statement. Each $p \in P$ is labelled with the function $\mathcal{L}_p : P \rightarrow \mathsf{expr}(V)$, where function expr is a variable expression on a subset of V; this corresponds to an `if` condition. Input labels make it possible to assign values to elements of V: $\mathcal{L}_i : I \rightarrow V$, which is mapped from a `receive` statement. By means of output labels it is possible to assign values represented by the $v_i \in V, 0 \leq i \leq |V|$ to an $o \in O$: $\mathcal{L}_o : O \rightarrow \times_k V_k$ on the analogy of a `send` statement. Action labels represent a transformation of values of V: $\mathcal{L}_a : expr(V) \rightarrow V$, these can be generated based on the assignments in the TTCN-3 test case source. Four mutually disjoint subsets of edges that correspond to TTCN-3 statement blocks are distinguished based on the types of nodes they connect: $T = T_{ss} \cup T_{sp} \cup T_{pp} \cup T_{ps}$. Each of these edges is labelled with a set of labeling functions defined in this section. The labeling function for an edge connecting two test states is defined as follows: $\mathcal{L}_{ss} : T_{ss} \rightarrow (S \times V) \times \mathcal{L}_o \times \mathcal{P}(\mathcal{L}_i \cup \mathcal{L}_a) \times (S \times V)$, where $\mathcal{P}$ is the set of all possible ordered subsets operator. The labeling function for an edge connecting a test state node and a predicate node is $\mathcal{L}_{sp} : T_{sp} \rightarrow (S \times V) \times \mathcal{L}_o \times \mathcal{P}(\mathcal{L}_i \cup \mathcal{L}_a) \times (P \times V)$. The labeling function for and edge connecting two predicate nodes is $\mathcal{L}_{pp} : T_{pp} \rightarrow (P \times V) \times \mathsf{expr}(V) \times \mathcal{P}(\mathcal{L}_i \cup \mathcal{L}_a) \times (P \times V)$ The labeling function for an edge that connect a predicate node and a test state node is $\mathcal{L}_{ps} : T_{ps} \rightarrow (P \times V) \times \mathsf{expr}(V) \times \mathcal{P}(\mathcal{L}_i \cup \mathcal{L}_a) \times (S \times V)$.

Figure 4 shows a TTCN-3 code snippet with a part of a test case definition on the left, and its graph representation after parsing on the right. Timeout events are represented with the name of the timer, which is T, and the parameterless `Q.receive` is denoted with an `*` input label. The test purpose of this test case is to check the overflow of a counter. It counts to the maximum value of a domain, for instance the size of a sliding window in the flow control mechanism of a protocol, then it checks if the overflow is handled correctly. The graph model has five nodes in the set S represented with circles in the figure, and three of them are labelled with the possible verdicts. There is one node $p \in P$ represented with a rectangle in the figure, and its label is the `if` condition: $s < w_s$. Edges are assigned with a set of labels. For instance the edge $e_3 \in T_{ss}$ from the unlabeled top node to the $\mathsf{inconc} \in S$ node is labelled with the name of the template sent ($t\_req \in O$) together with its parameters ($s \in V$), a timer action, and an input event ($T \in I$). Edge $e_4 \in T_{ps}$ from the $p \in P$ node to the unlabeled node at the top is labelled with a valid value for $\mathsf{expr}(p)$ that is in this example the boolean value `true`.

```
1  testcase tc_t (integer w_s)
       runs on C_CT{
2   map(mtc:Q,system:Q);
3   /* ... */
4   Q.send(t_req(s));T.start;
5   alt{
6     []Q.receive(t_resp(s,s+1)){
7      s := s+1;
8      if (s < w_s) {
9       Q.send(t_req(s));T.start;
10      repeat;
11     }
12    }
13    []Q.receive {
14     setverdict(fail)}
15    []T.timeout {
16     setverdict(inconc)}
17  }
18  Q.send(t_req(w_s));T.start;
19  alt{
20    []Q.receive(t_resp(w_s,0)){
21     setverdict(pass)}
22    []Q.receive {
23     setverdict(fail)}
24    []T.timeout {
25     setverdict(inconc)}
26  }
27 }
```

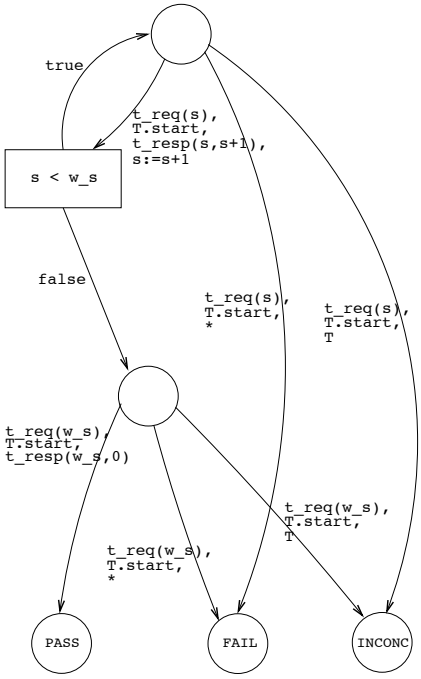

Fig. 4. Graph model of the test case. On the left a TTCN-3 code snippet with a part of a test case definition is shown, on the right its graph representation is shown

4 Mapping TTCN-3 Documents to a Relational Schema

Mapping data structures of different programming languages to database tables has been studied for more than two decades. Several frameworks have been proposed and developed, and some of these have made their way to the everyday practice. A widely used framework that implements the persistence of Java and .NET objects is for instance Hibernate [12][13], while Django [1] provides a framework for Python, and Ruby on Rails [2] gives a programming interface for Ruby.

The main idea of these solutions is that with a little piece of meta-information it is possible to make a one-to-one correspondence between the data types and database tables, data type attributes and database table attributes, and data type instances and database records. Association and aggregation relations

between data types and collection types are mapped by means of foreign keys in tables.

Former frameworks such as [14] intended just to hide the database back-end from the application. Our approach follows this track, because of the metadata necessary to enable the efficient tracking of changes of variable values and to support logging.

4.1 Mapping TTCN-3 Type Definitions to a Relational Schema

Several approaches can be defined to map a TTCN-3 type definition to a relational schema. Though TTCN-3 is not an object-oriented language, object-relational mapping should be considered: mapping record types to tables, record fields to table attributes, and values to the records of that table. In this case, the mapping from TTCN-3 to the database is very easy, however the reverse direction is far less straightforward as type definitions are available at runtime, and the meta information encoded in the schema may not be enough to reconstruct the value.

We use a mapping that preserves all meta-information. It maps data flow elements of a TTCN-3 document to a relational schema of six tables derived from the test data graph model. Figure 5 shows the entity-relationship diagram of the database schema we use for storing a TTCN-3 type definitions, and below we list these tables and their attributes within parentheses, where the attributes that define the primary key of the table are underlined.

- Datatype(name, type)
- Compositefield(parent, fieldname, typeof)
- Template(name, typeof)
- Templatevalue(template, field, value)
- Typeinstance(name, typeof)
- Typeinstancevalue(variable, field, simplevalue, parent)

The Datatype table stores each data type declared in the testcase. Its name attribute is the name of the data type, and the type attribute can take the following values: primitive, record. These values correspond to the primitive data types and the record type, respectively.

Each node of the data type graph of Fig. 3 is stored as a record in the Datatype table. The attributes of the record are the name of the data type, and primitive if the given data type has no child nodes in the type graph, otherwise the attributes are the type name and record.

The Compositefield table is responsible for storing the type hierarchy, more precisely, the types and field names of record type fields. Each record of this table corresponds to a field of a record type, that is, an edge in the graph of Fig. 3. The parent attribute stores the name of the type of the given field is declared in. The name attribute stores the name of the field, and the typeof attribute stores the type of the field.

The Template table holds information about the parameterized templates defined in the test source. Each of its records corresponds to a template, a bold

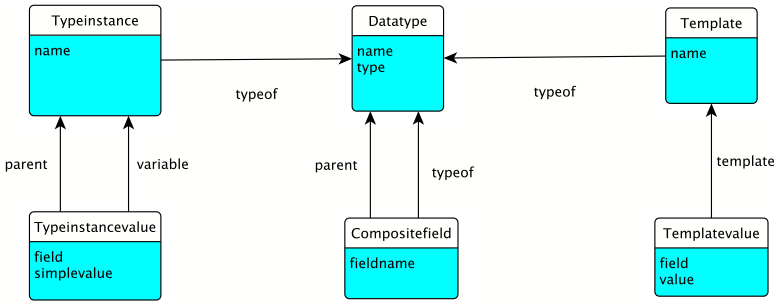

Fig. 5. Entity-relationship diagram for the metamodel of test data and test case used in the database

node in Fig. 3 with with no incoming bold edges. The name attribute stores the name of the parameterized template, and the typeof attribute, which is a foreign key to the Datatype table, stores the data type of the template.

The Templatevalue table stores parameterized templates as defined in the test code. Each of its records corresponds to a template field, to a bold edge in Fig. 3. The template attribute, which is a foreign key to the Template table, stores the name of the template containing the given field. The field attribute stores the name of the field, and the value attribute stores the value of that field.

The Typeinstance table is similar to the Template table. It stores the names and types of data type instances including variables and generated data type instances in its name and typeof fields, respectively.

The Typeinstancevalue table stores the field values of record type instances or the value of primitive type instances. Its attribute named parent stores the name of the parent data type instance of the given field. The field attribute stores the name of the field. If the parent instance is of a primitive type, the simplevalue attribute stores the value of the parent instance, and in that case the field attribute is an empty string, and the variable attribute is null. On the other hand, if the parent type is a record type, then the simplevalue attribute is null, and the variable attribute stores a reference to the data type instance storing the value of the field. The parent relationship is used in the next section when assigning this value to a variable.

As an example of the mapping described in this section, type `C` and template `t_c` are mapped to the following records of tables Datatype, Compositefield, Template and Templatevalue. In the example below, the Datatype table has two records that correspond to two of the types used, `C` and `integer`.

name	type
C	record
integer	primitive

The Compositefield table details the record type `C`, which has three fields, all of the type `integer`, named `t`, `s` and `r` that appear as records of the table.

parent	field	typeof
C	t	integer
C	s	integer
C	r	integer

As a template represents a value of a type available in the Datatype table, the Template table associates the template name and the type name, in this example `C` and `t_c`.

name	typeof
t_c	C

Template `t_c` is an instance of type `C`, hence it has tree fields. The Templatevalue table assigns the value of the variable identified with `v_t` to field `t` of template `t_c`, and similarly initializes the two other fields as well.

template	field	value
t_c	t	v_t
t_c	s	v_s
t_c	r	v_r

4.2 Random Template Generation with SQL

Gecse's method assumes that the relational logic should make it possible to generate random values of a type to be used when generating a template. We use the data structure shown in Fig. 3, that is used as the basis for the template generation. In the following, we show how these random values can be obtained using SQL queries. The generator algorithm is implemented in the Template Generator component of the relational access layer.

In the following queries, there are references to vector elements. These elements are not included in the queries in their presented forms, but are substituted with the values stored in the referenced vector element.

1. The first step of the random template or value generation is the collection of available data types, which can be obtained using the following SQL query:

   ```
   SELECT name, type FROM datatype
   ```

   The result of the query is stored in vector types in the memory.
2. Next, we iterate through types, and generate a number of random names for each type later to be used as identifiers. The names generated for the i^{th} element of types are stored in the vector at the i^{th} position of the vector names. Each type-name pair, in this case the i^{th} type and its j^{th} name, is

stored in the database using the following pseudo-SQL statement:

```
INSERT INTO typeinstance (name, typeof)
  VALUES (names[i][j],types[i])
```

3. Then, all these newly generated type instances are queried:

```
SELECT d.name typename, t.name instance, d.type
  FROM datatype d, typeinstance t
  WHERE d.name=t.typeof AND
   t.name NOT IN (SELECT parent FROM typeinstancevalue)
```

   The result of the above query is the set of all newly generated instance names. Since the Typeinstancevalue table can store some variable values before starting the generation, in order to get the list of newly generated instances, the records belonging to values have to be filtered from the list of all type instances. This is done by not including those instance names in the list, which are already included in the Typeinstancevalue table (the newly generated instances are not yet included in the Typeinstancevalue table so this selection condition is appropriate). The result of this query as a list of type name, instance name, type triplets is stored in vector typesnames in the memory.
4. The next step is to generate the values of each newly generated template name. This is done by iterating through typesnames and generating random values for each instance.
   If the current element of vector typesnames is of a primitive type, then a random value rand is generated according to the type of the instance, and inserted into the database by the following statement:

```
INSERT INTO typeinstancevalue
  (parent, field, simplevalue, variable)
  VALUES (typesnames[i].instance,'',rand,null)
```

   If the current element of this vector is of a record type, then its fields are queried by the following pseudo-SQL query in the i^{th} cycle:

```
SELECT field, typeof FROM compositefield
  WHERE parent=typesnames[i].typename
```

   The result of this query is a list of all the fields that the current type instance has according to its type, and it is stored in vector in the i^{th} position of the vector fields.
5. In the next step, we iterate through this vector, that is, the elements of i^{th} element of fields, and one randomly chosen, previously generated instance name is assigned to each field of the current instance. This is done by the pseudo-SQL query below, and the j^{th} random value is stored at the j^{th} position of the value candidate vector of the i^{th} field, which is an element at

the i^{th} position in the vector value:

```
SELECT name FROM typeinstance
  WHERE typeof=fields[i][j].typeof ORDER BY RANDOM() LIMIT 1
```

The above query selects the list of all instances generated for the type of the current field, which are then ordered randomly (`ORDER BY RANDOM()`), and only the first element of the so-created result is selected (`LIMIT 1`).

6. Finally, the result of this query is stored in the Typeinstancevalue table using the following pseudo-SQL statement:

```
INSERT INTO typeinstancevalue
  (parent, field, simplevalue, variable) VALUES
  (typesnames[i].instance,fields[i][j].field,null,value[i][j])
```

Thus, according to the semantics of the Typeinstancevalue table, the parent attribute gets the name of the instance the field value whose has just been selected. The field attribute gets the name of the current field. The simplevalue attribute is null, since the parent instance is a record. Finally, the name attribute gets the newly selected value.
According to the earlier described semantics of the Typeinstancevalue table, in the case of a type instance of a primitive type, the parent attribute gets the name of the parent type instance, the field attribute is an empty string, the simplevalue attribute is the value generated randomly according to the type of the instance and the variable attribute is null.

4.3 Mapping a TTCN-3 Test Case to a Relational Schema

The model of each test case is stored in a relational schema as well to enable the tracking and logging of test case execution, to be able to track the changes of variable values made by assignment, receipt of messages, and tackle the effect of predicates to the control flow.

Figure 6 shows the entity-relationship diagram of the database schema we use for storing a TTCN-3 test cases. The listing below shows the entities and their attributes within parentheses, and the primary key attributes are underlined, multiple underlined attributes denote a composite primary key.

- State(id, verdict)
- Transition(id, from, next, output, parameterlist, expression)
- Action(transition, serial, type, expression, input, value)

The State table stores information about states of the test case. For each state of the test case, one record is inserted into this table. The id attribute is the artificially generated identifier of the state as test states are unlabeled. The verdict is the verdict value that labels leaf states. For non-leaf states, the verdict attribute is null.

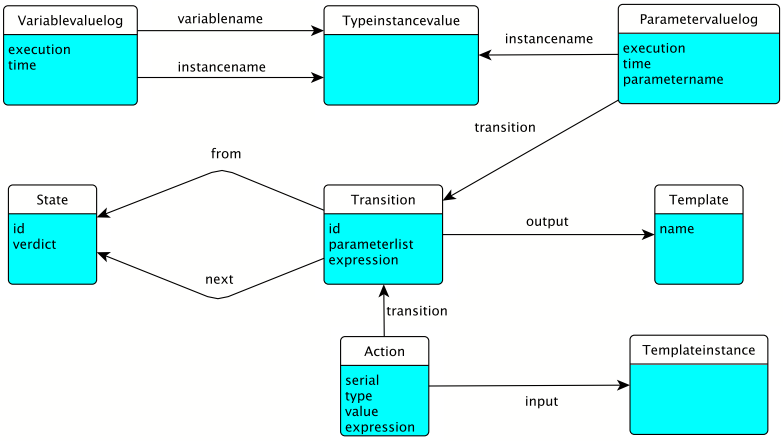

Fig. 6. Entity-relationship diagram for the metamodel of test data and test case used in the database

For each transition one record is inserted into the Transition table. The id attribute of a transition is its artificially generated identifier. The from and next attributes store the current and next test states of the corresponding transition, respectively, and these two attributes are foreign keys to the State table. The output attribute stores a reference to the template sent as output, and the parameterlist attribute stores the list of those variables, which are the parameters of that template. These two attributes are null in case of a predicate transition, i.e., a transition starting from a predicate denoted with a rectangular node in Fig. 4. For such transitions, the expression attribute stores the triggering expression, which is null for output triggered transitions originating in test states.

The Action table stores the input events and value assignments performed during state transitions, whose ordering is determined by the serial attribute. Its transition attribute refers to the transition, during which the action is performed, thus, it is a foreign key to the Transition table. The type attribute of the action can take two values: `assignment` and `input`. The former represents a value assignment, where an internal variable of the test case gets the value of an evaluated expression, while the latter represents a message input action. The expression attribute stores the expression that updates a variable in case of an assignment, while the value attribute, a foreign key to the Typeinstance table, stores the variable, to which the input value or the expression value is assigned in case of an input or assignment action, respectively. The input attribute that is a foreign key to the templatevalue table stores a reference to the input template. The following two functions define the mappings of action and input labels of the test case model to this relational schema.

Logging and making snapshots of test states are vital parts of our framework. For logging the test execution, two additional tables are defined that together can be used for tracing any test execution (see top of Fig. 6).

- Variablevaluelog(execution, time, variablename, instancename)
- Parametervaluelog(execution, time, transition, parametername, instancename)

The Variablevaluelog table stores the data type instances used as values of variables during test execution. Each record of this table belongs to one variable value assignment, where variable value assignment means selecting a value from the database and assigning it to a variable. The execution attribute of the Variablevaluelog table stores the identifier of the test execution, while the time attribute stores the timestamp of the assignment. The variablename attribute stores the name of the variable and the instancename stores the identifier of that randomly generated data type instance the value whose is assigned to the variable. The instancename and variablename attributes are both foreign keys to the Typeinstance table.

The Parametervaluelog table stores the values assigned to template parameters during test execution, each record of this table belongs to a parameter value assignment. The execution attribute of the table is the identifier of the test execution, during which the value assignment is performed. The time attribute is the timestamp of the assignment. The transition attribute stores that transition, during which the message with the value assigned parameter is sent. The parameternname attribute stores the name of the parameter, while instancename stores the identifier of the data type instance assigned to the parameter.

5 TTCN-3 Test Case Transformation and Execution

The role of test case transformation is to remove all data from the test case definitions, and to access all those pieces of data from the database through external functions:

- For each data type, a parameterized template definition is generated unless it already exists such that it has a parameter for each field of the type of the template. For type M in Fig. 3 the following template is constructed:

```
1 template M t_m (A v_a, C v_c, octetstring v_d) {
2   a := v_a,
3   c := v_c,
4   d := v_d
5 }
```

- Original template references are replaced with these parameterized template definitions. In the example we use in this paper, there is one parameterized template for each data type. Any template with a different parameter list is replaced with this most general template. For instance, if before the transformation a template definition for type `M` existed with a single parameter, it is replaced with the definition above.
- All variables, templates and formal parameters are redeclared as TTCN-3 `anytype`, however in the database these are still represented with their real type. The `anytype` is the only type in TTCN-3 that is able to represent all

possible types, therefore it can be used in the native layer to dynamically generate a value of the proper type. Note that this is removed from the examples for the sake of better readability. In the example of Fig. 4, the formal parameter variable `integer w_s` is transformed into `anytype w_s`, and its value is referred with `w_s.integer`.
- For each literal, a local constant is introduced in the test case body, that constant is initialized with that literal, and the value is saved by an external function to the database. In the example shown in Fig. 4, there is one literal: `1`. The local constant is introduced in `const integer c_ONE=1`, and it is stored with `var('c_ONE','integer','1')`.
- For each constant within the test case body, the value is loaded to the database, and the constant is initialized with an external function.
- For each expression, the expression is passed to the relational access layer as a charstring that returns with the evaluated value. For instance, `expr('i+1')` resolves the expression, replaces the `+` operator with its SQL correspondent, which is in this case `+`, and executes an SQL query that reads the value of the resolved variables and evaluates the expression. The expression in this case is translated to the following query:

```
SELECT simplevalue+1 FROM typeinstancevalue WHERE parent='i'
```

- An assignment is replaced with an external function call that pairs the name of the variable with the value of the right hand side expression. For instance, the `i := i + 1` of Fig. 4 is replaced with `set('i',expr('i+c_ONE'))`.
- A variable access such as in template parameters is replaced with a function call that returns with the proper value. For instance, template `t_c(c_ONE, r, s)` is substituted with `t_c(val('c_ONE'),val('r'), val('s'))`.
- After every `receive` operation, the template and its fields are stored in the database with a `set` external function call.

During test case execution, when a test case is initialized, the database, the log and snapshot are reset, variables are set to their initial values. Whenever a `send` or `receive` or `timeout` statement is activated or a database access layer function is called, some of the variables are accessed and database statements are executed. When a variable is accessed for the first time, its value is set for the rest of the test case. If it has not yet been defined, then we call that variable free, so it is possible to generate a value for that with Gecse's method described in section 4.2 possibly at random. If the variable has already been set, its value is changed based on the parameters of a `set` function call and its previous value.

The example in Fig. 4 is a TTCN-3 snippet from simplified version of an HDLC flow control. The test case checks if the incrementation of the received sequence number is correct after an overflow. As the value of variable `w_s` is removed from the test case, the same test case can be reused even for different protocol versions of this family.

The parameterization of a frame represented by template `t_m` to be sent still works identically to the original test case. However, as the value of the data field

`d` of type `M` is free in the whole test case, our framework makes it possible to generate its value on-the-fly, instead of using a hardcoded `'0'` value. The same statement holds for the template `v_a` at any point in the example, but the cycle variable `s` that represents the sent sequence number is bound after its first access.

At runtime, the data transformation module saves and fetches the `anytype` values from the memory to the database and vice versa. The easier direction, just like in the case of the object-relational mapping, is the saving of a memory object to the database. This requires only the parameterization and the execution of a few well-defined SQL statements. However, the reverse direction is more challenging. The value fetched from the database must be decoded into a variable of an arbitrary type known only at runtime, while decoders are available only for primitive TTCN-3 types at compile time. The name of that type is known at runtime, so the field of the `anytype` union to be set is known as well. If that type is a TTCN-3 record type, its fields are – recursively – initialized one-by-one in the native module. If that type is a primitive type, the value produced by the decoder is assigned to that field.

6 Conclusion

The framework presented in this paper increases the reusability of test case definitions at the cost of reducing the portability. With test data removed from the TTCN-3 documents, test parameterization takes place in a relational database back-end, and it is possible to test the behavior with different set of parameters. Our method makes the identification of template fields transparent for the executing test case possible, and can generate appropriate values for those fields.

This framework has been implemented as a prototype extension plugin for the TITAN TTCN-3 Test Executor [8]. The test case and test data models are extracted from the TTCN-3 code with a specialized parser. The execution framework that provides the on-the-fly database access consists of a large set of external functions implemented in C++. To make variable identification by variable name possible, we use metadata and C++ reflection mechanisms, and in the TTCN-3 documents all variables and templates are declared as `anytype`. This framework has been tested with SQLite 3 as database back-end.

Our experiments we made on Diameter Base Protocol [15] and two sample protocols show that the overhead of using a database is a little over 1 ms for each access on average. Though this value is small, this framework should not be used for performance testing.

References

1. Django: Django (2005-2013), `http://www.djangoproject.com/`
2. Hansson, D.H.: Ruby on Rails (2013), `http://rubyonrails.org/`
3. Baker, P., et al.: Data-Driven Testing. In: Model-Driven Testing, pp. 87–95. Springer (2008)
4. Soapui.org: Functional Tests – Data Driven Testing (2013), `http://www.soapui.org/Data-Driven-Testing/functional-tests.html`

5. Gallio: MbUnit (2012), http://gallio.org/wiki/doku.php?id=mbunit
6. EaseTech: easytest (2013), http://github.com/EaseTech
7. Fitnesse Test Manager: Fitnesse Resources (2010), http://fitnesse.testmanager.info/fitnesse/DataDrivenTesting.pdf
8. Szabo, J.Z., Csondes, T.: TITAN, TTCN-3 Test Execution Environment. Infocommunications Journal 57(1), 27–31 (2007), www.hiradastechnika.hu/data/upload/file/2007/2007_1a/HT_0701a-6.pdf
9. Gecse, R.: Towards automatic generation of a coherent TTCN-3 template framework. In: Núñez, M., Baker, P., Merayo, M.G. (eds.) TESTCOM 2009. LNCS, vol. 5826, pp. 223–228. Springer, Heidelberg (2009)
10. Tretmans, J.: Testing Concurrent Systems: A Formal Approach. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 46–65. Springer, Heidelberg (1999)
11. Wang, C.J., Liu, M.T.: Generating Test Cases for EFSM with Given Fault Models. In: Proceedings of the Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies, Networking – Foundation for the Future, vol. 2, pp. 774–781. IEEE (1993)
12. O'Neil, E.J.: Object/relational Mapping 2008 – Gibernate and the Entity Data Model (EDM). In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, pp. 1351–1356. ACM (2008)
13. JBoss.org: Hibernate (2013), http://www.hibernate.org/
14. Alashqur, A., Thompson, C.: O-R gateway – A System for Connecting C++ Application Programs and Relational Databases. In: USENIX C++ Technical Conference, pp. 151–170. Usenix Association (1992), http://openlibrary.org/works/OL12608637W/USENIX_C_Technical_Conference
15. Fajardo, V., Arkko, J., Loughley, J., Zorn, G.: Diameter base protocol (2012), http://tools.ietf.org/html/rfc6733