Refactorisation methods for TTCN-3

L. Eros, F. Bozoki

In this paper we introduce automatic methods for restructuring source codes written in test description languages. We modify the structure of these sources without making any changes to their behavior. This technique is called refactorisation. There are many approaches to refactorisation. The goal of our refactorisation methods is to increase the maintainability of source codes. We focus on TTCN-3 (Testing and Test Control Notation), which is a rapidly spreading test description language nowadays. A TTCN-3 source consists of a data description (static) part and a test execution (dynamic) part. We have developed models and refactorisation methods based on these models, separately for the two parts. The static part is mapped into a layered graph structure, while the dynamic part is mapped to a CEFSM (Communicating Extended Finite State Machine) – based model.

Keywords: TTCN-3, formal methods, refactorisation, automatic.

1 Introduction

Testing is becoming an increasingly important phase in the development process. The sooner a fault is found in the source code, the fewer resources it takes to correct it. Automating test cases significantly improves the efficiency and reduces the duration of testing. Many tools have been applied for testing purposes, for example TTCN-3 for automating whole test cases [1].

Refactorisation is a commonly used technique for changing the syntax of program codes without making any changes to their behavior [2], [3]. We have concentrated on refactoring TTCN-3 source codes.

1.1 Related work

Many tools have already been developed for refactoring sources written in different languages, such as C++ and Java [4, 5], and even for TTCN-3 [6]. These tools are all semi-automatic, which means that the developer has to interact during the refactorisation process. These semi-automatic tools aim easier readability of the source. We have concentrated on achieving easier maintainability, scalability, and a compact source. This kind of refactorisation can be carried out automatically, without human interaction.

There have not yet been any automatic tools for refactoring TTCN-3 sources, so our goal was to develop data structures and automatic algorithms for refactoring TTCN-3 sources.

1.2 Introduction to TTCN-3

TTCN-3 is a test description language that was standardized by ETSI in 2000 [1].

The test written in TTCN-3 runs on a test executor. The executor is connected to the SUT (System Under Test). From the viewpoint of TTCN-3, the SUT is a black box, that is, TTCN-3 determines whether the SUT works as it should by examining the responses given by the SUT for certain inputs.

A TTCN-3 source consists of **modules** on the topmost level. Each module has two parts, namely, the module definition part and the module control part [1].

The **module definitions part** includes declarations of data types, module-level variables, ports and definitions of templates. Most of the generally used **simple data types** (integer, char, charstring) can be found in TTCN-3, but it has **structured data types** as well (record, set) [1]. Templates are used for defining the structure of messages to be sent or received.

The **module control part** coordinates the test execution. It contains function calls, message sending and receiving instructions, and value notations [1].

2 Refactoring the static part

In this section we introduce a data model and an algorithm for refactoring the module definitions part. The data model consists of graphs that the data declarations and definitions of the source can easily be transformed to. The algorithm seeks for redundancy in this model and reduces it using inheritance (modified templates in TTCN-3) and references. We will concentrate on refactoring the definitions of record-typed templates.

2.1 Data model for the static part

First of all, the original TTCN-3 source has to be transformed into a data model by which the refactoring steps can be carried out efficiently. This model consists of two layers (Fig. 1). The lower layer is a directed graph called the type graph, while the upper one consists of directed trees called value trees.

The **type graph** consists of two kinds of nodes: T-nodes and F-nodes (Fig. 1). Each **T-node** represents a data type in the source. It stores the name of the data type that it represents, as well as pointers to its parent node and child nodes. As the structured data types in the source, each T-node has its fields in the data model, which are represented by the **F-nodes**, the child nodes of the T-node. An F-node stores the name of the field it represents as well as pointers to its parent node and child node, which is a T-node that represents the data type of the field. The **value trees** are built from the template definitions of the source code. They consist of **V-nodes** that store the values defined in the templates of the source. A V-node also contains pointers to its parent node and child nodes which are all V-nodes, and a pointer, the **modifies-pointer**, which is only set if the template represented by the tree inherits the values of some of its fields from another template.

Once the type graph and the value trees have been created, the value trees have to be connected to the type graph in the following way: Each V-node is connected to the T-node

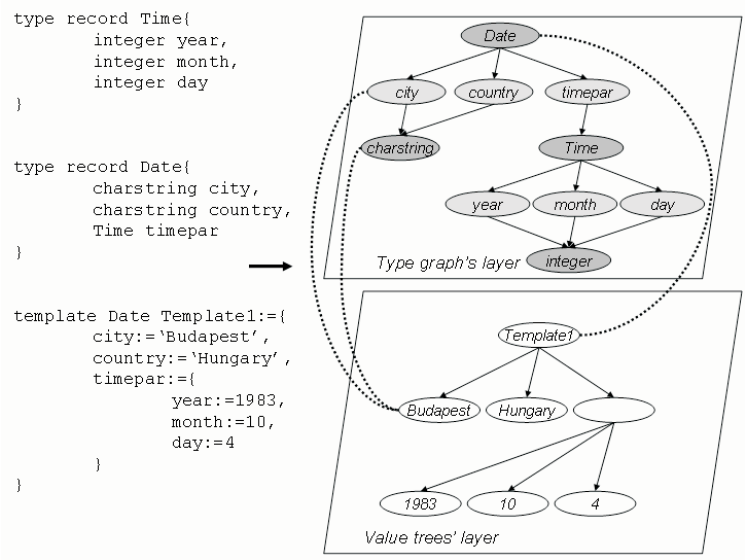

Fig. 1: Transforming the source into the data model (F-nodes: light grey, T-nodes: dark grey, V-nodes: white)

that represents the type of value stored in the V-node and to the previously mentioned T-node's parent node (if one exists), which is an F-node (Fig. 1).

2.2 Ways of refactorisation

We have concentrated on two types of redundancy. In the following we will write about these two kinds of redundancy, and the ways in which our algorithm reduces them.

The first type of redundancy is caused by **equal templates**, in other words, templates of the same type with all the corresponding fields having equal values. In this case, the algorithm uses references to reduce the redundancy. There are two cases of this type of redundancy.

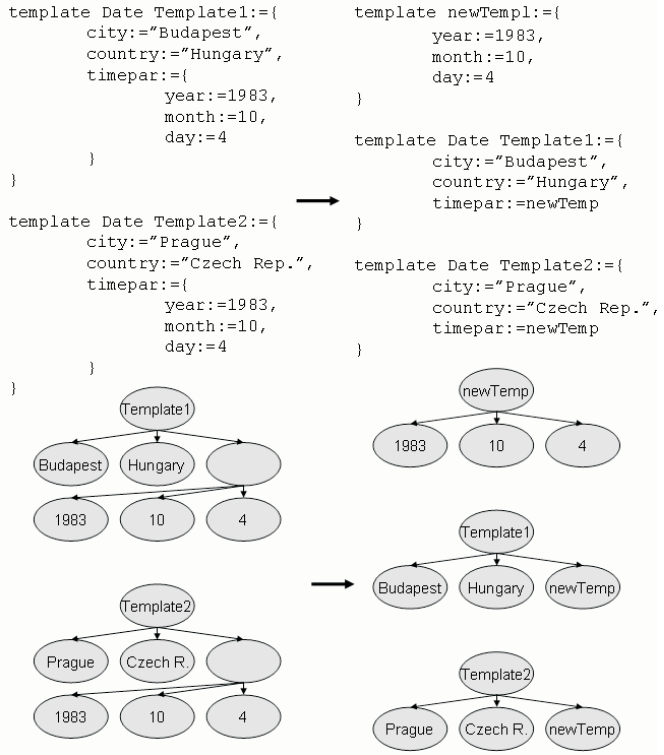

Fig. 2: Refactorisation by reference

In the one case, a separately defined template appears as a part of another template. Handling this case is simple: the repetitive sub-template has to be replaced by a reference to the separately defined template.

In the other case, a template that was not defined separately appears several times as a sub-template of other templates. When handling this kind of repetition, the repetitive sub-template has first to be defined separately, then all of its occurrences have to be replaced with a reference to this newly defined template. An example of this kind of refactorisation in the source and in the model can be seen in Fig. 2 (the original structures are on the left, while the refactorised ones are on the right).

The second type of redundancy is caused by **similar templates** of the same type, in other words, templates that have relatively many identical fields. This kind of redundancy is handled by modified templates (inheritance) in the following way: one of the similar templates has to be left just as it was before, and the other has to be turned into a modified template that only redefines the non-equal fields and inherits the rest from the other template (in the data model the modifies-pointer has to be used). Fig. 3. shows how this kind of refactorisation works in the source and in the data model.

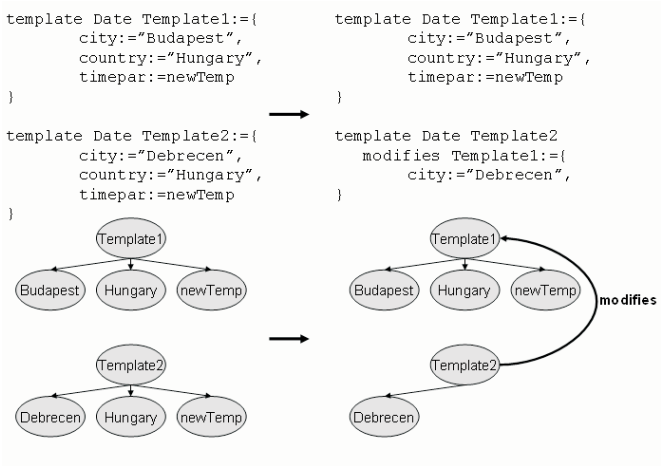

Fig. 3: Refactorisation by inheritance

To be able to manage the level of similarity between two value trees of the same type, we defined the **RoS** (Rate of Similarity):

$$RoS = \frac{\text{number of equal leaves}}{\text{number of leaves of the value tree having more leaves}}. \quad (1)$$

If the *RoS* of two value trees exceeds a limiting value, these two value trees should be refactorised.

2.3 Refactoring algorithm of the static part

In this section, we introduce the refactorisation algorithm of the static part. The algorithm has four steps as follows.

In the **first step** of the algorithm, the **N-matrices** are created for each T-node. These matrices are used for determining if two value (sub-) trees of the same type are equal. An element of the matrix is 1 if the corresponding two value trees are equal, if not, it is 0.

In the **second step**, the equal value (sub-) trees of the same type are handled by traversing the N-matrix of the T-node. The N-matrix is traversed twice. During the first traverse, the algorithm only handles the repetitions where one of the value trees is a standalone tree, and during the second traverse it handles all the remaining repetitions (the repetitions where both of the value trees are sub-trees of other value trees). This ensures that as many repetitions as possible are handled by references to originally defined value trees.

In the **third step**, the **D-matrices** are created for each T-node. These matrices store the RoS for each pair of value trees. After creating the matrices, the values below the limiting value of the T-node are cleared.

In the **fourth step**, the D-matrices are used to create maximal weight spanning trees with the value trees as their nodes, for each T-node, using Prim's algorithm [2]. Then two value trees are refactored by inheritance if they are connected by an edge in the spanning tree. The reason for building a maximal spanning tree is that in this way a maximal number of fields can be defined by inheritance, so a minimal number of fields have to be redefined.

3 Refactoring the dynamic part

In this section we introduce a data model and an algorithm for refactoring the module control part. The main point of this algorithm is to find repetitive sequences of instructions and turn them into bodies of functions or altsteps (special functions in TTCN-3), depending on their structure [1]. The original occurrences of the repetitions are replaced by calls of the corresponding functions.

3.1 Data model for the dynamic part

The module control part is transformed into a CEFSM-based model. A CEFSM is represented by a directed graph. In our approach, a CEFSM state consists of a node and a directed edge in this graph. The states have several attributes:

The **guard** is a condition that enables the transition to the state, the attribute **event** is the event that leads to the transition to the state, the **action list** is the sequence of instructions to be executed during the transition between two states, while the attribute called **parameters** contains the parameters of the action list.

When mapping the source into the CEFSM-model, the **receive** and **timeout** instructions of the source will be the events of the CEFSM graph and the instructions between two receive instructions will be instructions of action lists. However, some TTCN-3 structures need to be handled in different ways.

An **if-else structure** is mapped into a two-armed branch. One of the arms gets the if-condition as its guard and the instructions between the if- and the else-statement, while the other arm gets the negative of the if-condition as its guard and the instructions after the else-statement. Both of the events are empty. An example of this kind of transformation is shown in Fig. 4.

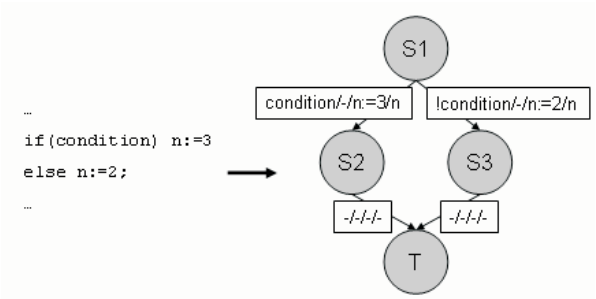

Fig. 4: Transforming an if-else structure into the CEFSM-model

Alt structures describe alternative behavior [1]. These statements are also mapped into branches. An alt-branch has the same number of arms in the model as the alt statement has in the source. The guards and events of the arms in the source will be the guards and events of the arms in the

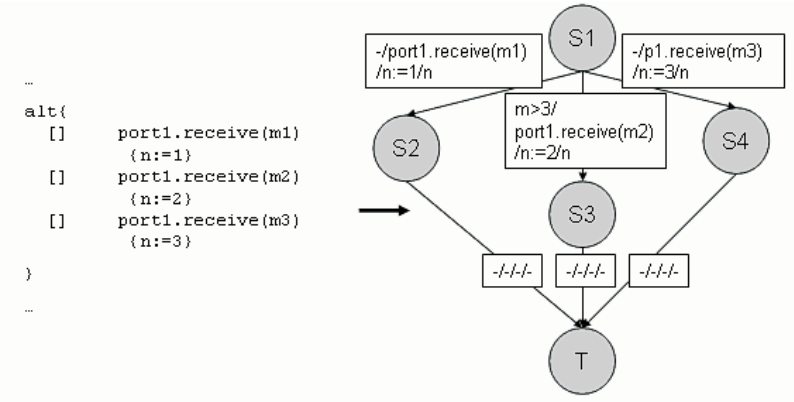

Fig. 5: Transforming an alt structure into the CEFSM-model

CEFSM. The action lists will contain the instructions of the corresponding arms in the source code (Fig. 5).

The states labeled "T" in the above figures are called **terminating states**. They are used for collecting the arms of the branches. All their attributes are empty.

Functions, **for-cycles** and **while-cycles** also have to be handled in special ways, since – when turning a sequence of instructions into the body of a function or an altstep – instructions from inside the body of a function or a cycle must not be handled together with instructions from outside it. To avoid these kinds of cases, we have introduced **hyper states**. A hyper state looks like a simple state from the outside, but it contains the CEFSM-model of the body of the function or cycle inside. The event of a hyper state is also special. It can be *call_for*, *call_while*, or *call_function*, depending on the kind of structure that it represents. Thus, functions and cycles are transformed into hyper states, and function calls are transformed into states referencing the corresponding hyper state (Fig. 6.).

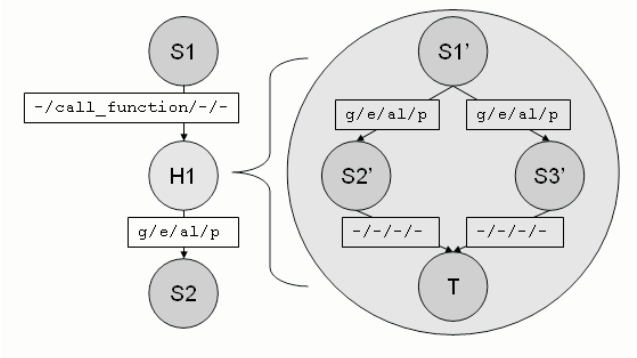

Fig. 6: Hyper state representing a for-cycle in the CEFSM-model (g=guard, e=event, al=action list, p=parameters)

3.2 Pre-arranging the action lists

After creating the CEFSM-model, the action lists have to be pre-arranged into a uniform order in order to create potentially longer repetitive sequences of instructions in the action lists. This pre-arrangement is possible, because the order of some instructions can be changed without changing the behavior of the system, and it is useful, because handling these longer repetitive sequences eliminates more redundancy. The rules of this pre-arrangement are as follows:

Changing the order of message sending instructions is not allowed. If referencing a variable, the last value notation of the variable has to be kept before the referencing instruction. The declaration of the variable has to come before the first reference to the variable and before all the instructions that change its value.

To achieve this uniform order, a dependency graph is generated from the instructions of the action list. Each instruction is mapped to a node. If a directed edge points from node A to node B, then the originating node (the instruction corresponding to node A) must be kept before the terminating node (the instruction corresponding to Node B). Fig. 7 shows an original source code, its dependency graph, and the source code after the pre-arrangement.

3.3 Handling sequential repetitions

This step of the algorithm searches for repetitive sequences of instructions within the action lists. Before starting to seek for these repetitions, list **AL** has to be created with the action lists as its elements, beginning with the longest one. After creating AL, the search for repetitions works as follows: The algorithm selects a sequence, called **BS** (base sequence) that all the sequences with the same length are compared to. At the beginning, the length of BS is equal to the length of the longest action list, then it is decreased by one in each iteration. Since the same sequence is not likely to be found in the same action list, once the algorithm has found a sequence that matches BS, it jumps to the next action list (the next element of AL). When all the sequences that match BS are found, a whole repetition is explored. If sequences of the newly found repetition overlap with sequences from repetitions that were found earlier, they are thrown away. If the **size** of the repetition (the product of the length of BS and the number of its occurrences) are below a limiting value, the repetition is thrown away. Finally, the repetition is stored in list **RL**, which contains some information about the repetitions found, and a new BS is selected. If the repetition contains equal action lists, this is indicated in matrix **RM**, which is used in the next step of the algorithm. When RL is complete, the repetitive sequences are turned into functions.

3.4 Handling structural repetitions

After handling the sequential repetitions, the algorithm searches for repetitions having repetitive structures that cover more states. Two states are **equal** if their events and action lists are equal. The input of this part of the algorithm is RM. In each iteration of this step, the algorithm chooses a pair of

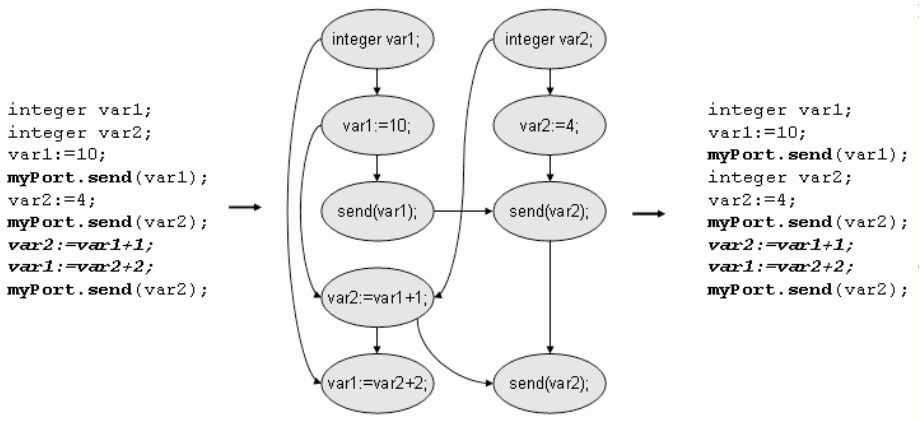

Fig. 7: Original code, dependency graph and rearranged code

states having action lists that were equal according to RM. If the events of these two states are equal (the states are equal), then their sibling nodes in the CEFSM-graph are compared. If all the corresponding siblings are equal, then their parent nodes are examined in the same way, recursively, until the whole repetition is revealed. Then the whole repetition is turned into the TTCN-3 structure (function or altstep) that best fits the structure of the repetition [1].

4 Case study

In this section we demonstrate our algorithm on a TTCN-3 source code. The original and refactorised source codes are shown in Fig. 8.

```
module example{

type record infType{
        integer postal_code,
        charstring street,
        integer house_num,
}

type record reqType{
        integer year,
        infType address
}

type record respType{
        charstring housetype,
        infType address,
}

template respType address:={
        postal_code:=1025,
        street:="Jozsefhegyi",
        house_num:=2
}

template reqType request:={
        year:=2007,
        address:=address
}

template respType response:={
        house_type:="condominium",
        address:={
                postal_code:=1025,
                street:="Jozsefhegyi",
                house_num:=2}
}

type port port1 message {
        out reqType;
        in  respType;
}

type component MTCType {
        port port1 c_p;
}

testcase Testcase1() runs on MTCType {

        map(mtc:c_p, system:c_p);
        timer T1;
        T1:=3;
        T1.start;
        c_p.send(request);

        alt{
          [] c_p.receive(response){
                verdict.set(pass);}
          [] T1.timeout{
                verdict.set(fail);}};

        T1:=3;
        T1.start;
        c_p.send(request);

        alt{
          [] c_p.receive(response){
                verdict.set(pass);}
          [] T1.timeout{
                verdict.set(fail);}};
        stop
}

control{execute (Testcase1())}}
```

```
module example{

type record infType{
        integer postal_code,
        charstring street,
        integer house_num,
}

type record reqType{
        integer year,
        infType address
}

type record respType{
        charstring housetype,
        infType address,
}

template respType address:={
        postal_code:=1025,
        street:="Jozsefhegyi",
        house_num:=2
}

template reqType request:={
        year:=2007,
        address:=address
}

template respType request:={
        house_type:="condominium",
        address:=address
}

type port port1 message {
        out reqType, infType;
        in  respType;
}

type component MTCType {
        port port1 c_p;
}

function_1(inout timer T1){
        T1:=3;
        T1.start;
        c_p.send(request);
}

function_2(inout timer T1){
        function_1(T1);
        alt{
          [] c_p.receive(response){
                verdict.set(pass);}
          [] T1.timeout{
                verdict.set(fail);}};
}

testcase Testcase1() runs on MTCType {
        map(mtc:c_p, system:c_p);
        timer T1;
        function_2(T1);
        stop
}

control{execute (Testcase1())}}
```

Fig. 8: Original and refactorised TTCN-3 source codes

5 Comparison with other solutions

In this part we focus on a refactorisation tool for TTCN-3 named T-Rex [6]. The basic concept of this solution is to search the source for typical patterns called smells, indicating that the quality of the code can be increased (for example, a smell can be an unused parameter). The list below shows some of the smells that are implemented in T-Rex (in practice, many smells are not implemented):

- Constant Actual Parameter Value smells for templates,
- Duplicate Alt Branches,
- Fully-Parametrised Templates,
- Singular Component Variable/Constant/Timer Reference,
- Singular Template reference.

The method searches for these kinds of smells and indicates them to the user, who can decide to ignore or correct the indicated smells. Unlike our method, T-Rex is a semi-automatic approach as user interaction is needed for the refactorisation. This solution rather focuses on the readability of the code, while our goal is to decrease its redundancy and increase its maintainability.

6 Summary

In our paper we have introduced data models and algorithms for refactoring source codes written in TTCN-3. The data model of the module definitions part is a layered model. It consists of the type graph and the value trees. The algorithm uses references and inheritance in order to reduce the redundancy in the module definitions part. The module control part is transformed into a CEFSM-model. In this model the algorithm seeks for sequential and structural repetitions and turns them into functions or altsteps. In this way, the source becomes more compact and more easily maintainable and scalable.

Acknowledgments

First of all, we would like to thank our supervisors, Gyula Csopaki, Ph.D. and Antal Wu-Hen-Chang for their direction and for their advice during our research. We would also like to thank our department for providing the necessary equipment to us.

References

[1] ETSI ES 201 873-1 3 1.1 *Methods for Testing and Specification (MTS) The Testing and Test Control Notation Language, version 3; Part 1: TTCN-3 Core Language*, ETSI, 2005.

[2] Weiss, M. A.: *Data Structures and Algorithm Analysis in C++*, Addison-Wesley, 2006, p. 373–376.

[3] Mens, T.. Tourwe, T.: Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, Vol. **30** (2004), No. 2, p. 126–139.

[4] Ref++ for refactoring C++ sources, http://www.refpp.com

[5] Transmogrify for refactoring Java sources http://transmogrify.sourceforge.net

[6] Neukirchen, H., Bisanz, M.: Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites, *TestCom /Fates 2007 conference*.

Levente Eros
e-mail: el492@hszk.bme.hu

Ferenc Bozoki
e-mail: bf490@hszk.bme.hu

Dept. of Telecommunications and Media Informatics

Budapest University of Technology and Economics
Magyar Tudosok korutja 2
1117 Budapest, Hungary