A New Approach in Model-Based Testing: Designing Test Models in TTCN-3

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

[1] Department of Telecommunications and Media Informatics, Budapest University of Technology and Economics, Magyar tudósok körútja 2, H-1117, Budapest, Hungary
{adamis,eros,kovacsg}@tmit.bme.hu

[2] Ericsson Hungary
Irinyi J. u. 4-20, H-1117, Budapest, Hungary
{antal.wu-hen-chang,tibor.csondes}@ericsson.com

Abstract. Throughout the years, many model description languages have been used in different model-based testing tools, however, all these languages are quite unfamiliar to test engineers. In this paper, we propose TTCN-3 (Testing and Test Control Notation 3), the nowadays most popular and widely spread test definition language to be used for this purpose and give two alternative approaches how this could be carried out. TTCN-3 as modelling language can support test generation tools by means of the annotations we introduce in the paper.

Keywords: TTCN-3, extended finite state machine, model-based testing.

1 Introduction

When black box testing a communicating system, the test environment does not have any information about the internal structure of the System Under Test (SUT) and it can only investigate the SUT through the outputs it replies for different inputs. Nowadays, TTCN-3 (Testing and Test Control Notation 3) language is widely used for defining black-box tests [1], and many test designers have become familiar with TTCN-3.

Manual test design and testing however, has many drawbacks, since the whole testing process is carried out by humans. With the aim of getting rid of these problems, model-based testing, which is a partly automatic test generation method, has become popular in the last years. In the case of model-based testing, the test designer does not have to implement the test manually. Instead, he or she creates an abstract model of the SUT based on which, test cases are generated automatically, by a model-based testing tool. By transferring a part of the task of test generation from humans to automatic methods, model-based testing gains many advantages, as we will see in Section 2.

In the last two decades, several model-based testing tools have been developed. Recently, three of the most promising tools are Conformiq Tool Suite, Elvior TestCast Generator , and SpecExplorer by Microsoft [2,3,4,5]. These tools have different theoretical background and use different modelling languages for describing the abstract model of the SUT. In the case of Conformiq Tool Suite, the model of the SUT is described by UML (Unified Modeling Language) statecharts [6], and Java compatible source codes. Based on these inputs, Conformiq is capable of generating test scripts in various formats, like C/C++, Java, TTCN-3, etc. For Elvior TestCast Generator , the model of the SUT has to be described by UML state machines, and the tool generates TTCN-3 test cases as its output. SpecExplorer uses Abstract State Machine Language (AsmL) [7] and Spec# [8] for modelling, and as its output, it generates a data structure called a transition system, which can be converted to a C# code for running the test.

Besides the above, several other tools, languages and modelling techniques have been used for model-based testing. Tools Lutess [9], Lurette [10], and GATeL [11] use Lustre [12] as the language for modelling the SUT. Lustre is a declarative, synchronous dataflow language. In the case of Conformance Kit developed at KPN Research and PHACT developed at Philips, the model of the SUT is specified by an Extended Finite State Machine [13] (see in Section 2). TVEDA [14] and AutoLink [15] use SDL (Specification and Description Language) [16] for modelling the SUT, while in the case of TVEDA, Estelle [17] can be used as the modelling language as well. Cooper [18] uses LOTOS (Language of Temporal Ordering Specifications) [19,20] as its input language, while in the case of TorX [21], the SUT can be modeled by LOTOS, Promela [22], and FSP (Finite State Process) [23].

These tools use different test derivation methods for generating test cases from the abstract model of the SUT. GATeL uses constraint logic programming (CLP) for generating test cases from the model [24]. Conformiq Tool Suite, Elvior, Conformance Kit, PHACT, AutoLink and TVEDA generate the necessary test cases based on finite state machines (FSM) [13].SpecExplorer, Cooper, and TorX use Labeled Transition Systems (LTS) [25] based theory for test case generation.

This paper investigates whether it is possible and practical to use TTCN-3 as modelling language and as the foundation for model-based testing. The rationale of adapting TTCN-3 relies on the following facts. It is a popular test definition language with tool and compiler support, and for many test engineers this is the only language they use on an everyday basis. Therefore, test designers who have skills in TTCN-3 do not need to learn a new formal description technique or language. TTCN-3 developers are likely to have designed many TTCN-3 test data structures of protocols, which exist in a data structure library and are ready to be used in test generation tools as well. This paper thus attempts to merge the advantages of the popularity of both TTCN-3 and model-based testing, by proposing, investigating, and comparing different approaches of utilizing TTCN-3 for describing the abstract model of the SUT. When defining the abstract formal model to TTCN-3 mappings, we took the experience of other existing formal description techniques and model-based testing approaches into account.

Partial TTCN-3 support has been alraeady available for modelling purposes in Elvior TestCast Generator and Conformiq Tool Suite. The former has native TTCN-3 support for data type and template type definitions, moreover, it has a feature that allows referring TTCN-3 functions in the UML model. The latter provides a TTCN-3 type definition import feature. This paper goes beyond this level of integration, and regards TTCN-3 as an exclusive modelling language.

The rest of the paper is organized as follows. In Section 2, we review the goals and basic concept of model-based testing. Section 3 presents our approaches for creating Extended Finite State Machine (EFSM) models and illustrates those through the example of the Alternating Bit Protocol. Section 4 takes a look at the advantages and disadvantages of the methods. An annotation-based extension of the TTCN-3 language is given in Section 5 that can make test generation based on TTCN-3 models more effective. Finally, a brief summary is given.

2 Model-Based Testing

There are multiple ways of black box testing communicating systems. In this section, we are going to present the partly or fully manual ways of test generation methodologies, the problems they raise [26], and the ways in which model-based testing attempts to solve these problems [26, 27].

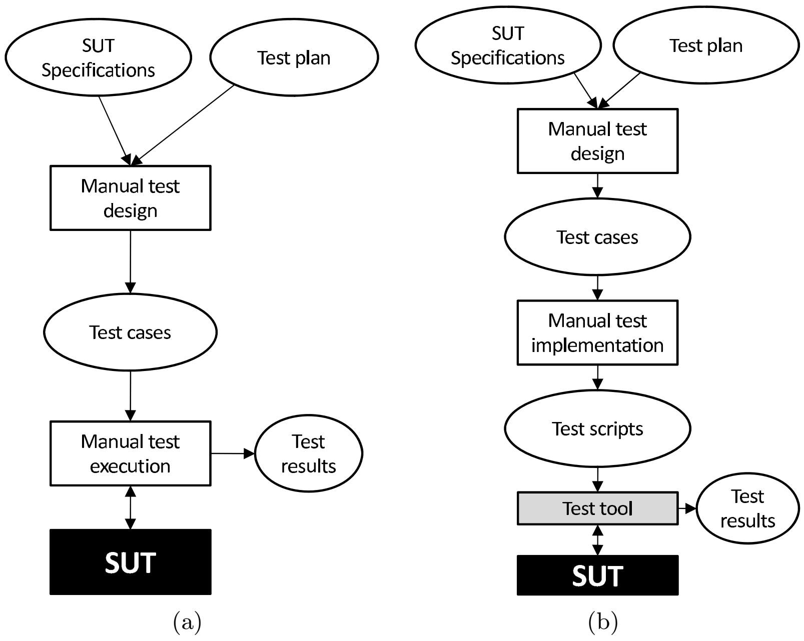

Fig. 1. Manual (a), and script based (b) test design

Figure 1(a) shows the steps of manual testing. When manually developing test cases, the test designer uses as input the written, informal or formal specifications of the SUT, and a test plan, which includes the purposes or goals of the test. Based on these inputs, the test designer develops the test cases manually on a high level of abstraction. The test cases created are then passed from the test designer to the tester. Following the steps described in the test cases, the tester executes the test manually by interacting with the SUT, and records the outputs the SUT returns for the inputs of the test case.

The so-called script based tests speed up the process of testing by automating the test execution phase. As in Figure 1(b), the test designer does the same job as in the manual case that is, designs the necessary test cases based on system requirements and the test plan. The tester on the other hand, rather than manually executing the test based on the high-level test cases, implements a test script. The created test script that interacts with the SUT, that is, it sends different inputs to the SUT, and observes the outputs sent by the SUT. Based on these observations, the script generates a test verdict as its output.

Although script based testing makes test execution a lot easier than in the case of fully manual testing, it is unable to solve the further problems raised by manual testing. Thus, tests remain unstructured, their coverage is incomplete, the number of necessary test cases can be huge, and thus, they are hard to maintain. In other words, the test is designed in an ad-hoc way, the quality

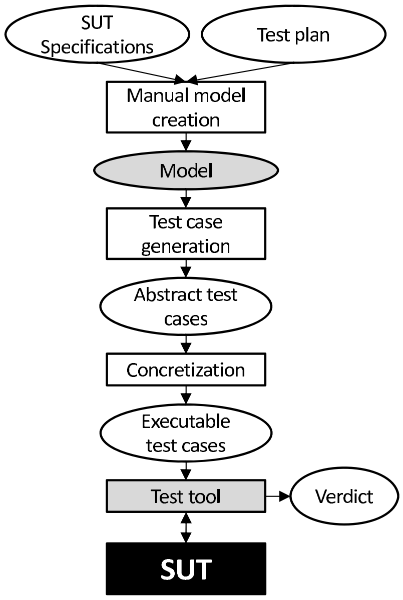

Fig. 2. Model based test design

of test depends on the expertise of the test designer. Besides, the design and maintenance of test cases is time consuming and costly.

The main goal of model based testing is to eliminate the problems mentioned above by partly automating test case generation. As Figure 2 shows, in model based testing instead of explicitly designing the different test cases to be run on the SUT, the test designer creates a *model* of the SUT. This model is written on an abstract level omitting some details of the behaviour of a valid SUT.

This paper focuses on finite state machine (FSM) based models, however, the model of the SUT can be given in a large variety of other formal modelling techniques. Finite state machines have been employed for modelling sequential circuits in hardware design, lexical analysis in compiler theory and pattern matching in artificial intelligence. Definition 1 gives a possible interpretation of the deterministic Mealy machine [13].

Definition 1. *An FSM is a quintuple: $FSM = (I, O, S, \delta, \lambda)$, where I and O are the finite sets of input and output symbols, respectively, S is the finite, nonempty set of states, $\delta : S \times I \rightarrow S$ is the state transition function, and $\lambda : S \times I \rightarrow O$ is the output function.*

Using an EFSM, defined in Definition 2, the values (state) of some variables are separated from the S control state set of the machine. Each EFSM can be unfolded to an FSM, but – depending on the size of the domain of variables – the number of states (elements of S) of the FSM can be by orders of magnitude larger than the number of states (elements of S) of the corresponding $EFSM$.

Definition 2. *An Extended Finite State Machine (EFSM) [13] is a quintuple $EFSM = (I, O, S, V, T)$, where I and O are the finite, nonempty sets of input and output symbols, respectively, S is the finite, nonempty set of states, V is the finite set of variables, and T is the finite set of transitions. Each transition $t \in T$ is a 6-tuple $t = (s_t, q_t, i_t, o_t, P_t, A_t)$, where $s_t \in S$ is the current state, $q_t \in S$ is the next state, $i_t \in I$ is the input, $o_t \in O$ is the output, $P_t(V)$ is a predicate on the current variable values, and $A_t(V)$ is an action on variable values.*

After designing the model, a test generator tool is used for automatically generating abstract test cases – traces – from the model. The test engineer usually has control over the testing efforts and number of test cases to be generated. This way, the theoretically infinite number of test cases can be limited.

The main benefits of model based testing are the following. The time and cost of test case generation is reduced, because the large number of necessary test cases is generated by the model based testing tool instead of a human. Also, the effect of human errors is reduced, since the test designer only creates a concise, abstract model, which is easier to see through than a manually written, long test code. Thus, model based test generation results in better test coverage and maintainability.

3 Designing Abstract Models in TTCN-3

This section takes a look at different approaches for defining – extended – finite state machine models in TTCN-3. In this paper, there are two alternative approaches, a functional one and a declarative one, considered that differ in the way of describing the behaviour, the system interface definition is the same in both cases. The functional approach defines the system behaviour as a large test case. The declarative approach defines a data structure that defines the constraints for maintaining the state of the EFSM and determining the appropriate transition function on an incoming event. The two approaches are illustrated on the example of the Alternative Bit Protocol.

3.1 Modelling System and Its Interfaces

First we introduce two refinements to the EFSM model given in Definition 2. The input set I and the output set O can contain huge numbers of events. To reduce their sizes, messages with similar structures are considered to be parameterized, we allow $i \in I$ and $o \in O$ be parameterized with a variable list $V' \subseteq V$ in our model, and denote them with $i(V)$ and $o(V)$. The second refinement is that we allow the input/output alphabet to be partinitioned into finite number of sub-alphabets such that $I = I_1 \cup I_2 \cup \ldots \cup I_N \cup \iota$ and $O = O_1 \cup O_2 \cup \ldots \cup O_N$, where ι is an internal event, $1 \leq N < \infty$ and I_i and O_i may be empty.

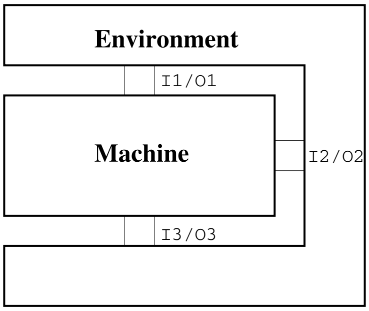

Fig. 3. EFSM interfaces

The abstract model of EFSM interfaces is shown in Figure 3. The model divides the domain into two parts, the machine itself and its environment. In the figure, the input and output sets of the machine are partitioned into three subsets, the communication between the machine and the environment takes place by message-based interaction through these ports.

An EFSM can be modelled with a TTCN-3 module. The module definition consists of two main parts, the interface definition part and the behaviour definition part. The interface definition contains optionally data type definitions and definitions of templates for output messages of the machine, at least one port type definition, and the definitions component types of the machine called `SystemType` and its environment called `EnvironmentType`.

For the parameters of each $i(V) \in I$ and $o(V) \in O$ a TTCN-3 data type is defined if it does not yet exist. For each sub-alphabet $I_i/O_i, 1 \leq i \leq N$ a message-based port type is defined such that for all $i_{ij}(v_j) \in I_i, 1 \leq j \leq |I_i|$ and $o_{ij}(v_j) \in O_i, 1 \leq j \leq |O_i|$ in port type i the type of v_j is declared as an inout parameter. For each $o(v) \in O$ a parameterized template type is defined with a format parameter of the type of v. For all $v \in V$, $i(v) \in I$ and $o(v) \in O$ a variable is declared in the component type of the system corresponding to machine M, and a port of each port type is declared in both component types. These mappings are summarized in Table 1.

Table 1. EFSM to TTCN-3 model – interface mappings concepts

EFSM element	**TTCN-3 element**
EFSM	Module, Component type definition
Set of variables	Data type definition
Input/Output message parameters	Data type definition
Input/Output set partition	Port definition
Output message	Parameterized template

Example. The interface definition of machine M is illustrated on the simple Alternating Bit Protocol. The EFSM model of this protocol can be seen in Figure 4. The protocol provides the reliable transmission of user data to a peer entity. The receipt of user data is guarded by a one-bit sequence number that alters on each successful transmission. In the current example user data is considered to be an integer value, and ports are named after the input or output messages received or sent through them. The example is taken from [28].

The protocol has two control states ($Idle$ and $Wait\_Ack$) and four transitions. The system is initially in the $Idle$ state. On input of user data on the usr port of the system in the $Idle$ state, the user data is coupled with the current value of the one bit sequence number and transmitted via the snd port, and the state machine begins to wait for an acknowledgement. In the $Wait\_Ack$ state, three transitions are possible. On no input message on the rcv port of the system a timeout event (ι) occurs resulting in the retransmission of the previous user data value coupled with the current sequence number. If a sequence number is received on the rcv port, it is compared to the current sequence number value. If the two are the same, the system remains in the $Wait\_Ack$ state. If different, the current value of the sequence number is inverted, and the state machine is brought back to the $Idle$ state.

Formally, the elements of the EFSM tuple are the following. The set of states S is $\{Idle, Wait\_Ack\}$. The set of variables V has two element NS and $userData$. The input and output alphabets are $I = \{usr, rcv, \iota\}$ and $O = \{snd\}$ respectively. I is partitioned into $\{usr\} \cup \{rcv\}$. The usr input message has an integer parameter, the rcv input message a Boolean parameter and the parameter of

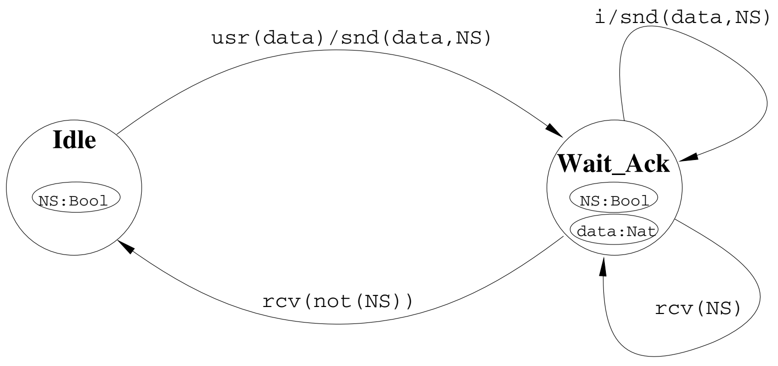

Fig. 4. EFSM of the Alternating Bit Protocol [28]

the *snd* output message is a record composed of an integer value and a Boolean value. There is an internal message ι to the system itself, this message models a timeout event.

The format of each message in the input/output alphabet is defined by a data type. The data types used in our example are shown in the listing below. The `AlternatingBit` is a `bitstring` limited to two possible values. The `DataFrameType` is a record composed of an `AlternatingBit` value representing the counter and and `integer` value representing the user data. As user data are considered to be `integer` values there is no need to define a new type for that.

```
type bitstring AlternatingBit ('0'B, '1'B);
type record DataFrameType {
  integer data,
  AlternatingBit counter
}
```

Output messages are represented with parameterized templates. The role of the template is to construct the data structure from its fields and initialize optional fields. In case of the Alternating Bit Protocol, there is only one outgoing message, the data frame sent via the `snd` port. The data record of that message has two fields, so the template must have two parameters, one of `integer` type, the other of `AlternatingBit` type.

```
template DataFrameType tDataFrame (integer pl_d,
    AlternatingBit pl_b) :={
  data:=pl_d,
  counter :=pl_b
}
```

The EFSM model of the Alternating Bit Protocol defines three ports for communicating with the environment: the `usr` port towards the user of this protocol entity, the `snd` port for sending data to the peer entity, and the `rcv` port for receiving acknowledgements for messages sent from the peer entity. As these ports are shared by the system and the environment, the values that can be offered through them are declared as `inout` parameters.

```
type port USRPortType message {
  inout integer;
}
type port SNDPortType message {
  inout DataFrameType;
}
type port RCVPortType message {
  inout AlternatingBit;
}
```

The TTCN-3 module representing the EFSM model has two component types: the system itself and its environment as shown in Figure 5.

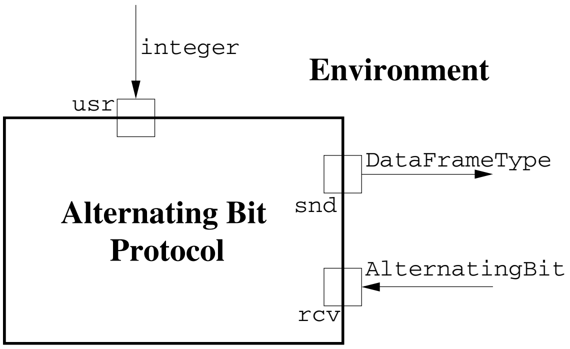

Fig. 5. Ports of the Alternating Bit Protocol

The component type of the system must declare besides the port definitions a variable for each element of the V set of the EFSM and each outgoing message parameter value. For each timeout event in I a timer is declared, in the code below the timer called `T_retransmission` corresponds to the inobservable input event ι.

```
type component SystemType {
  var AlternatingBit v_counter;
  var AlternatingBit v_received;
  var integer v_userData;
  timer T_retransmission := 5.0;
  port USRPortType usr_PT;
  port SNDPortType snd_PT;
  port RCVPortType rcv_PT;
}
```

The component type of the environment contains only the port definitions as shown below.

```
type component EnvironmentType {
  port USRPortType usr_PT;
  port SNDPortType snd_PT;
  port RCVPortType rcv_PT;
}
```

3.2 Functional Approach for Modelling Behaviour in TTCN-3

The behaviour of the EFSM is determined by the transition function according to Definition 2. This means formally that the elements of the $T \subseteq (S \times V) \times (I \times P(V)) \times (A(V) \times O) \times (S \times V)$ product.

The TTCN-3 language elements that can take part in the behaviour modelling are `control`, `testcase`, `if..else`, `alt`. The role of `control` is to execute the single `testcase` of the module. The role of the `testcase` is to implement the behaviour. This `testcase` is however turned inside out, it runs on the `SystemType` component type and its `system` is the environment.

A state $s \in S$ represents an execution point in the body of the `testcase` that can be represented with a `label`, a `function` name or an `altstep` name. In most cases the `testcase` contains an infinite loop that iterates over these points. The set of variables V is inherited from the component type the testcase runs on.

A transition is triggered by an $i \in I$ input event and selected by the $P_i(V) \subseteq P(V)$ set of conditions. At each execution point denoted as state all input events of I are allowed to be received and can be selected with an `alt` statement. The enabling conditions of the transition are checked with a set of `if..else` constructs.

The result of the transition is a set of outputs $O_i \subseteq O$ and a set of changes in the variable set V defined by $A_i(V) \subseteq A(V)$. An output event o_i is mapped to a `send` statement that takes a template parameterized with $V_i \subseteq V$ as argument. The variable set V is changed once by the parameters of the input event and updated with the `values` construct, and updated with the actions $A_i(V) \subseteq A$ during the transition by means of assignments.

The next state is set at the end of the executing transition by means of a `goto` if states are represented with labels, or setting the name of the next function or altstep to be executed in the body of the test case by means of a function reference.

The behaviour of the test case can be further decomposed into functions or altsteps, which can be parameterized with a subset of the ports and variables of the system and can therefore be used for introducing hierarchical considerations in the model.

Table 2 summarizes the functional approach for mapping EFSM transitions to TTCN-3 language elements.

Example. The mapping of the functional approach is illustrated on the example of the Alternating Bit Protocol introduced in the previous section.

Table 2. EFSM to TTCN-3 model – Functional approach for mapping behaviour

EFSM element	TTCN-3 element
State	`label` or `function` name or `altstep` name
Input	`alt` and `receive` and `values`
Predicate	`if..else` construct
Output	`send` using a parameterized template
Action	assingment
Next state	`goto` or function reference

```
control {
  execute (tc_Behaviour());
}
```

In the test case that is executed the modelled system type is declared as `mtc` and the environment type is declared as `system`, this means that these are swapped compared to real test cases. The ports of the `mtc` and `system` are connected as in test cases, and since all ports declare one `inout` value, the mapping is symmetric. The behaviour is specified in an infinite loop.

```
testcase tc_Behaviour (AlternatingBit pl_c := '1'B)
     runs on SystemType system EnvironmentType {
  v_counter := pl_c;
  map (mtc:usr_PT, system:usr_PT);
  map (mtc:rcv_PT, system:rcv_PT);
  map (mtc:snd_PT, system:snd_PT);
  var boolean v_r := true;
  while (v_r) {
    label idle;
    usr_PT.receive (integer:?) -> value v_userData;
    snd_PT.send (tDataFrame(v_userData, v_counter));
    T_retransmission.start
    goto waitAck;
    label waitAck;
    alt {
      [] T_retransmission.timeout {
        snd_PT.send(tDataFrame(v_userData, v_counter));
        T_retransmission.start
        goto waitAck
      }
      [] rcv_PT.receive(AlternatingBit:?) -> value v_received
          {
        if (v_received==v_counter) {
          repeat;
        }
        else {
            T_retransmission.stop;
            goto idle;
        }
      }
    }
  }
  stop
}
```

3.3 Modelling Behaviour with State Transition Tables

Alternatively, EFSM behaviour can be described in a declarative way as well. Just like in the case of the functional approach the `control` executes the single `testcase` of the module that runs on the `System` component type as `mtc` and the enviroment as `system`.

The mapping in this case is done by introducing data types for storing states, inputs, transition events and next states of the machine. A map data type with the key of state name and input event name pair and the value of function names defines the set of transition functions. Formally $I \times S \to F$ is used, which means that the name of a state and the input event serve as a key to select the appropriate TTCN-3 function to be called with a function reference.

Example. The code below shows a piece (the transition triggered by the internal event that retransmits the user data) of the data structure of a set of state machines that implements the Alternating Bit Protocol. Variable `state` is the current state of a machine, and the `events` structure defines the state transition table. The `eventId` field represents an incoming event, and the `stateId` field of the `states` record represents a state. The `actions` field is a list of identifiers referring to functions to be called on the incoming `c_event_T_retransmission` event, if the machine is in the `c_state_waitAck` state.

```
v_fsms := { {
    state := c_state_idle,
    events := { {
        eventId := c_event_T_retransmission,
        states := { {
            stateId := c_state_waitAck,
            actions := {
                f_trans_fromWaitAck_toWaitAck_T_retransmission}
            }, ...
        }
    }, ...
    }
} }
```

The TTCN-3 function $f \in F$ associated with s and i implements $P(V) \times A(V) \times O \times S$ part of the transition, that is, it contains `if..else` constucts, assignments, `send` events and the setting of the next state name key. The code below shows the function that implements the transition selected in the example above.

```
function f_trans_fromWaitAck_toWaitAck_T_retransmission(
  in integer fsmId, in integer eventId, in integer stateId)
  runs on SystemType {
  snd_PT.send(tDataFrame(v_userData[fsmId], v_counter[fsmId])
      );
  T_retransmission.start
  v_fsms[fsmId].state := c_state_waitAck;
}
```

The name of the current state $s \in S$ is maintained in a variable of the component and the input event $i \in I$ is selected in the infinite loop in the body of the test case that listens for all possible events in an `alt` statement.

4 Analysis of Mapping Alternatives

The functional approach suits well to the concept of TTCN-3, it is easy to overview the code and therefore this method is practical for test developers. If components are implemented with processes or threads, the operating system could struggle with the large number of instances because of the frequent context switching. Between different instances only message-based communication is possible as internals a hidden from each other.

The advantage of the declarative approach is that the same behaviour can be instantiated multiple times making this approach suitable for performance testing. The number of instances is limited by the available memory. The instances run on the same component and can therefore share global variables. The disadvantage of this method is that it is difficult to define the data structure programmatically.

Table 3 compares the functional and the declarative approaches from several aspects. The declarative approach may have a better performance when dealing with multiple instances of the same behaviour at the cost of more difficult maintenance and a somewhat limited TTCN-3 language. Both approaches can support non-determinism by means of random number based – guarding – conditions. The TTCN-3 code used in both approaches is not only syntactically, but semantically valid.

Table 3. Comparison of the mapping approaches

	Functional	**Declarative**
Performance testing	no	yes
Maintenance	easy	hard
Expression Power	complete TTCN-3	limited TTCN-3
Non-determinism support	yes	yes
TTCN-3 code validity	semantic	semantic

5 Propositions

In order to produce the necessary test cases, beyond defining the abstract model of the SUT, the test designer has to provide the testing tool with some additional settings and parameters for the compiler. This can be done in configuration files, however, some parameters are model specific, so should be included in the model. While for instance, state space exploration depth should remain in the configuration file, the racing condition between two events is a vital part of the model. When such a consideration is used, test cases may become capable of handling racing conditions, so it can be possible to consider any order of reception of multiple messages sent by the SUT.

A possible way for adding such notations to the TTCN-3 code is the TTCN-3 Documentation [29]. However, we found the specialization of test cases beyond its capabilities. Thus, this section proposes annotation techniques for controlling racing conditions and test data selection.

To support test data generation, the possible values for the fields of templates considered during test generation may be given. The test designer should be able to define the minimum (`min`) and maximum (`max`) values of each template field, together with a `step` value to guide the equivalent partitioning technique. In this case, the test generator tool generates one template in which the value of that field is *min*, another template in which, the value of that field is *max*, and templates in which, the value of that field is $min + k * step$, where $k = 1, \ldots, \lfloor\frac{max-min}{step}\rfloor$. Furthermore, the test designer can define values (`forced_value`) that appear as field values in the generated templates. In our notation, forced values appear in curly brackets after an `@` symbol followed by the corresponding keyword in a comment that is placed in the line of the field value of the corresponding incoming template definition as the following example shows:

```
template OutputTemplateType output1 (integer a, charstring b)
    := {
  field1 := a //@min{15} @max{30} @step{10}
  field2 := b //@forced_value{"foo","bar"}
}
```

A racing condition can arise when the SUT sends multiple messages to the test environment sequentially via different ports, and the order in which the these messages are observed in the environment is not necessarily the same in all executions. Thus, the model based testing tool generates all possible orderings in `interleave` statement when this annotation is found. The test designer should inform the test generator tool about which messages of the SUT race with each other by labelling them. In our notation, each racing message *m* is labelled in curly brackets after an `@` symbol and a `race_label` keyword in a comment placed in the line of the `send` statement. The messages that race with that *m* message are referred to by their labels listed in curly brackets after an `@` symbol and a `races_with` statement placed in a comment in the line of the `send` statement. This is shown in the following example:

```
P1.send(output1); //@race_label R1 @races_with{R2,R3}
P2.send(output2); //@race_label R2 @races_with{R1,R3}
P3.send(output3); //@race_label R3 @races_with{R1,R2}
```

6 Conclusions

This paper proposed to use TTCN-3 as modelling language in model based test generation tools. This way test engineers would not need to learn a new notation for creating models beside the language they use every day in their work. Another benefit is that the valuable data structure libraries developed throughout the years in testing departments can be reused for modelling purposes. This way the time requirement and cost of model design could be significantly reduced and the quality of the model could be improved.

In this paper we suggested two alternative ways for designing abstract models of the SUT in TTCN-3. The functional approach follows the concepts of the TTCN-3 language and is practical for test developers. Though the declarative approach that stores the FSM transitions in a data structure is more difficult to maintain, it is suitable for performance testing as well. We proposed a syntax to guide model based test generation tools to be able to create more elaborated test cases that take for instance racing conditions and test data selection into account.

In the future we are going to extend our work on the annotation of TTCN-3 data structures with regard to functional dependencies between data structure fields. The work on TTCN-3 as modelling language can be further extended with hierarchical design and using parallel components within the model itself and a more sophisticated non-determinism support.

References

1. ETSI: ETSI ES 201 873-1 V4.2.1: Methods for Testing and Specification (MTS), The Testing and Test Control Notation version 3, Part 1: TTCN-3 Core Language (2010)
2. Conformiq: Tool suite, `http://www.conformiq.com/`
3. Huima, A.: Implementing Conformiq Qtronic. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 1–12. Springer, Heidelberg (2007)
4. Elvior: TestCast Generator, `http://www.elvior.ee/motes/`
5. Microsoft: Specexplorer, `http://research.microsoft.com/en-us/projects/SpecExplorer/`
6. Group, O.M.: Unified modeling language, `http://www.uml.org`
7. Microsoft: Abstract state machine language, `http://research.microsoft.com/en-us/projects/asml/`
8. Microsoft: Spec#, `http://research.microsoft.com/en-us/projects/specsharp/`
9. du Bousquet, L., Zuanon, N.: An overview of lutess - a specification-based tool for testing synchronous software. In: Proc. 14th IEEE Intl. Conf. on Automated SW Engineering, pp. 208–215 (1999)

10. Raymond, P., Nicollin, X., Halbwachs, N., Weber, D.: Automatic testing of reactive systems. In: Proceedings of the IEEE Real-Time Systems Symposium, RTSS 1998, p. 200. IEEE Computer Society, Washington, DC (1998)
11. Marre, B., Arnould, A.: Test sequences generation from LUSTRE descriptions: GATEL. In: Proceedings of the 15th IEEE International Conference on Automated Software Engineering, p. 229. IEEE Computer Society, Washington, DC (2000)
12. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language LUSTRE. Proceedings of the IEEE, 1305–1320 (1991)
13. Lee, D., Yiannakakis, M.: Principles and methods of testing finite state machines – a survey. Proceedings of the IEEE 84(8), 1090–1123 (1996)
14. Clatin, M., Groz, R., Phalippou, M., Thummel, R.: Two approaches linking test generation with verification techniques. In: Proceedings of the 8th International Workshop on Protocol Test Systems, IWPTS 1996 (1996)
15. Koch, B., Grabowski, J., Hogrefe, D., Schmitt, M.: Autolink- A Tool for Automatic Test Generation from SDL Specifications. In: Proceedings of Workshop on Industrial Strength Formal Specication Techniques (WIFT 1998), Boca, October 21-23, pp. 21–23 (1998)
16. ITU-T: Recommendation Z.100: Specification and Description Language (2000)
17. 9074, I.: Information processing systems – Open Systems Interconnection – Estelle: A formal description technique based on an extended state transition model (1989)
18. Alderden, R.: COOPER - The Compositional Construction of a Canonical Tester. In: Proceedings of the IFIP TC/WG6.1 Second International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, pp. 13–17. North-Holland Publishing Co., Amsterdam (1990)
19. ISO/IEC: ISO-880: LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behavior (1989)
20. van Eijk, P.H.J., Vissers, C.A., Diaz, M. (eds.): The Formal Description Technique LOTOS. Elsevier Science Publishers B.V., Amsterdam (1989)
21. Tretmans, J., Brinksma, E.: Côte de resyste: Automated model based testing. In: Schweizer, M. (ed.) 3rd PROGRESS Workshop on Embedded Systems, pp. 246–255. STW Technology Foundation, Utrecht (2002)
22. Holzmann, G.J.: Tutorial: Design and validation of protocols. Tutorial Computer Networks and ISDN Systems 25, 981–1017 (1991)
23. Magee, J., Kramer, J.: Concurrency: state models & Java programs. John Wiley & Sons, Inc., New York (1999)
24. Apt, K.: Principles of Constraint Programming. Cambridge University Press, New York (2003)
25. Tretmans, J.: Specification based testing with formal methods: A theory. In: Fantechi, A. (ed.) FORTE/PSTV 2000 Tutorial Notes, Pisa, Italy, October 10 (2000)
26. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach, 1st edn. Morgan Kaufmann, San Francisco (2007)
27. Pretschner, A., Philipps, J.: Methodological issues in model-based testing. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 281–291. Springer, Heidelberg (2005)
28. Saloña, A.A., Vives, J.Q., Gómez, S.P.: An introduction to LOTOS (1993), `http://www2.cs.uregina.ca/~sadaouis/CS872/lotos_language_tutorial.ps`
29. ETSI: ETSI ES 201 873-10 V4.2.1: Methods for Testing and Specification (MTS), The Testing and Test Control Notation version 3, Part 10: TTCN-3 Documentation Comment Specification (2010)