Generating TTCN-3 Test Cases from EFSM Models of Reactive Software Using Model Checking

Juhan Ernits$^{1,3}$, Andres Kull$^2$, Kullo Raiend$^2$, Jüri Vain$^{1,3}$

$^1$ Dept. of Comp. Sci. Tallinn Univ. of Technology, Raja 15, 12618 Tallinn, Estonia
vain@ioc.ee
$^2$ Elvior, Mustamäe tee 44, 10621 Tallinn, Estonia
{andres.kull|kullo.raiend}@elvior.ee
$^3$ Inst. of Cybernetics, Akadeemia tee 21, 12618 Tallinn, Estonia
juhan@cc.ioc.ee

Abstract: The paper describes a full procedure of executable test code generation from specification models using the Uppaal model checker. Test cases are generated for black-box testing of reactive software that is connected to the tester via an asynchronous message-based interface. For specifying the observable behaviour of the software under test we define a modelling language that is based on extended finite state machines. A model in such a language is transformed to a Uppaal model taking a structural coverage criterion as a parameter. Uppaal is used to find an abstract test sequence that is suboptimal in terms of length. Next, we present the rules for transforming the abstract test sequences to TTCN-3.

1 Introduction

The paper describes a full sequence of actions for automated test case generation in TTCN-3 for black-box testing of reactive software where software under test (SUT) interacts with its environment over asynchronous message-based interfaces. Applications of this kind of software can be found in, for example, telecom and web-service applications. Finite state machines and their extensions provide a convenient way of modelling such applications. We use extended finite state machines (EFSMs) to model the behaviour of the SUT. EFSMs have gained wide acceptance in software modelling [Ho02] and are used in the underlying model for specification languages like Statecharts [Ha87] and UML2 state machines [OMG03]. We use EFSMs as the starting point for the reason that they provide a semantically well-defined model representation that can be applied to test generation. The main value of our approach lies in the complete procedure of transforming a formal SUT specification into executable test code satisfying some selected structural coverage criteria.

In our approach the test cases are generated off-line, i.e. test cases are generated from the SUT model before the tests are run. We assume that the SUT model is deterministic, i.e.

$^*$ This work was partially supported by the Estonian Science Foundation under grant No 5775 and by ELIKO Competence Centre project ”Integration Platform for Development Tools of Embedded Systems”.

241
the input and expected output sequences are preset. We also assume that the model is strongly connected, that is, each state can be reached from any other state of the model (if necessary via reset transitions).

We generate test sequences from EFSM models using a guided model checker Uppaal Cora [BLR04]. The approach allows the user to specify various structural test coverage criteria of EFSMs, for example, selected states/ transitions, all transitions, all transition pairs etc. From the SUT model and a coverage criterion we construct a Uppaal model to achieve test sequences satisfying the criterion. The problem of generating test sequences is formulated as a bounded reachability problem and solved by model checking. When a model checker solves a reachability task it generates a witness trace that corresponds to an abstract test sequence. This abstract test sequence is further encoded to tester code in the TTCN-3 language [Gr03]. We selected TTCN-3 as the output because it is a dedicated language for testing purposes, it is standardized, and it is widely accepted in the software testing industry.

2 Related work

Different formal techniques like theorem proving [HNS97], constraint logic programming [Pr01], model checking and symbolic execution [HB94] can be applied to automated test case derivation. In recent years model checking has become more popular for test derivation due to a high level of automation. As stated in [FHP02] the most common coverage criteria in the context of model-based testing are structural coverage criteria. Test generation according to a structural coverage criterion is treated as a reachability problem and solved either by symbolic or explicit state model checking.

We are interested in minimizing the lengths of test sequences. Existing model checkers generate minimal-length witnesses typically by breadth-first search of state space that is known to be an NP-hard problem and thus calls for deeper insight. Optimization techniques used in model checking include pre-processing of the model, for example, cone of influence reduction and a step further is combining model checking with other constraint solving methods using scriptable model checkers as reported in [HMR04]. Instrumenting the model with trap variables is a standard technique used in prioritized traversal [BI04]. We use trap variables for specifying the reachability condition and combining guiding of Uppaal Cora with iterated search refinement [Er06] that is controlled by the test generation procedure.

Uppaal has been applied to testing in special testing environments like Uppaal Tron [La05] and Uppaal CoVer [He04]. Encoding the test coverage criteria as specialized built-in search strategies makes Uppaal CoVer efficient in generating deep test sequences. Uppal Tron supports on-line testing where the environment model serves as use cases that guide through the functionality of a particular interest. The cost automata-based tool Uppaal Cora [BLR04] is designed for solving guided reachability problems and can also be used for introducing context information to guide test case generation.
3 Workflow of TTCN-3 test case generation

The workflow comprises the following steps:

1. The test engineer defines the model of external behaviour of the SUT in the form of an EFSM. The SUT model is derived from the available specifications. In our approach the SUT model can be drawn by a CASE tool that is capable of drawing EFSMs (state charts) and exporting them into XMI. The currently supported tools include Poseidon 4.1 and Enterprise Architect 6.0.
2. The test engineer defines the interface of the SUT – input and output message types, message equivalence classes and interface ports. SUT interface is defined directly in TTCN-3.
3. The test engineer prepares the test data – instances of SUT input messages that the test cases will use to stimulate SUT during test execution. The test data is defined in TTCN-3, which allows the tool to import the data into the generated TTCN-3.
4. The test engineer defines the test configuration (test component definitions and port mappings in the case of more than one test component) directly in TTCN-3.
5. The test engineer defines a test coverage criterion for the test case generator.
6. The test case generator transforms the EFSM model into a Uppaal model and decorates the model with control information according to the selected coverage criterion. Guiding information for the model checker is added to the model.
7. The test case generator uses Uppaal Cora to find an abstract test sequence in the form of a sequence of transitions.
8. The test case generator converts the abstract test sequence to TTCN-3 modules. The generated TTCN-3 modules are executable in TTCN-3 test environments.

4 SUT model format

We use the definition of an EFSM from [LY96]. An EFSM is a finite state machine extended with context variables, actions (assignments) and guards containing these variables. In our approach the EFSM representing the SUT is drawn using a UML case tool.

To specify the EFSM we use the UML 2.0 state machine specification [OMG03]. For the purpose of test generation the general UML notation of transitions is constrained as defined in Figure 4.1. As the available tools do not support the UML 2.0 meta model fully, we utilize these elements with constrained syntax.

As the UML 2.0 state machine does not natively support context variables and message type definitions, we define context variables and message types in the SUT interface and test configuration specification in a separate TTCN-3 file. Timer actions are used to start and stop timers. Each timer is referred to by its unique identifier. The timer identifier is optional if there is only one timer in the model. Input messages are received from, and output messages are sent to, a particular port. Ports are defined by the user in TTCN-3. Output messages in actions refer to the corresponding message templates specified in TTCN-3. For generating non-blocking test cases an additional parameter max_delay
should be defined in conjunction with the sending of output messages. The max delay parameter defines the maximum time interval the tester should wait for the SUT output message before it can detect SUT non-conformance to the model. The default value of max_delay is used when it is not explicitly defined on a transition.

![BNF Syntax of Transition Label](image)

**Figure 4.1:** Syntax of the transition label in BNF

We use an example of the INRES protocol specified as an EFSM [Bo97] in Figure 4.2.

![EFSM Diagram of INRES Protocol](image)

**Figure 4.2:** Model of the INRES protocol as an EFSM [Bo97]

### 5 Using Uppaal to generate abstract test sequences

We support structural test coverage criteria of the model, for example, selected states, selected transitions, all transitions et al. For the purpose of test sequence generation we decorate the transitions of the model with trap variables according to the selected criterion. The trap variables are initially set to FALSE and they become TRUE one by
one when the relevant transitions are passed by the model checker. The model checker solves the reachability task to find a sequence of transitions that makes all trap variables TRUE. The abstract test sequence is produced by Uppaal Cora as a witness trace.

5.1 Transforming EFSM models to Uppaal

The transformation of the EFSM model to Uppaal comprises four steps (Figure 5.1). In the first step we reduce the search space of the model in a way that makes trace generation by model checking feasible. Thus, all receipt of input messages and transmittal of output messages are abstracted away from the model. Ommiting them does not affect the resultant witness trace produced by Uppaal but reduces the search space. From the input messages we preserve only the input message fields that are used in the assignments or guards and therefore influence the control flow of the model. Such input parameters are transformed to the Uppaal model as initialized context variables. Note that one has to use the same message field values in creating the input message instances.

In the second step the model is decorated with trap variables to mark passing certain states or transitions. Trap variable declarations and assignments are added according to the selected coverage criterion. In the third step the reachability task is encoded into an observer automaton.

![Figure 5.1: Transformation of EFSM model to Uppaal model for test generation](image)

In the fourth step cost functions are added to each transition according to the selected coverage criterion. Cost functions are used in Uppaal Cora to guide the model checker to search for traces that minimise the total resultant cost. We are interested in finding a trace that is suboptimal in terms of its length. The length of the test sequence is important due to the time taken by executing the test cases.

The simplest way to achieve all transitions criterion is to set an equal cost for each transition. The overall cost function is the sum of costs over all transitions. Uppaal Cora tries to find a trace that keeps the overall cost to a minimum and in the best case it finds a trace that passes all transitions just once. It is possible to work out more sophisticated algorithms to guide model checking using cost functions but these algorithms require some analysis of the model.
5.2 Generating abstract test sequences with Uppaal Cora

The techniques used for abstract test sequence generation from an EFSM model used in the current approach are described in detail in [Er06]. We apply iterated search refinement for generation of a suboptimal test sequence. The key idea is to use the bitstate hashing state space reduction method. Iterated use of such space reduction, combined with the random best depth first search of Uppaal Cora, yields test sequences that are considerably shorter than when generated by depth first search, and provides answers in cases where depth first search runs out of memory.

6 Transforming abstract test sequences to TTCN-3 test cases

Test cases are generated in TTCN-3 that can be executed as a tester against the SUT. An example of the resultant TTCN-3 is presented in Figure 6.1. Tester is a reflection of the SUT model. It can be modelled using the similar state machine as SUT but with inverted inputs and outputs. Tester evolves from state to state according to the observed SUT output and updates its own copies of context variables. Each transition in the tester side is encoded as a TTCN-3 function. The function sends a message instance from the message class that is specified in the SUT model as an input event of the transition. After this the tester waits for an expected output message as specified in the SUT model. If there are context variable updates specified in the model, the tester updates them according to the SUT model. When the tester receives an expected output message, the function returns successfully. If an expected output message does not arrive during max_delay the function throws setverdict(FALSE), meaning the observed behaviour violates the specification.

An abstract test case generated by Uppaal Cora is converted to TTCN-3 as a sequence of calls to the corresponding transition functions described above. The final statement of a TTCN-3 test case is setverdict(TRUE). Reaching this statement during test case execution means that the test case has been executed successfully and the tester has not observed SUT behaviour deviating from that expected by the model.

7 Conclusion

We presented a full cycle approach to generating test cases in TTCN-3 from specification models using the Uppaal model checker. Test derivation using model checking has two bottlenecks, which are both addressed: model construction and scalability. The model that is passed on to the model checker is generated automatically based on the selected coverage criterion and specification model. During transformation, cost functions are added to guide the model checker to obtain lengthwise suboptimal test sequences. The abstract test case is found using an iterated search refinement method that makes it possible to scale up the models and obtain sub-optimality of sequence lengths. The abstract test sequences are then encoded into TTCN-3 for execution in a generic test execution environment.
module AllTransitionsExample {

// Type definitions
    type charstring TsduElement;
    type record of TsduElement Tsdu;
    type record sendrequest{};
    type record datarequest{(Tsdu sdu, integer n, integer b)};

// Test data definitions
    template sendrequest sendrequest_l := {};
    template datarequest datarequest_l := {sdu := {"v[0]","v[1]"},
                                        n := 2, b := 2 };

// Communication port type definitions
    type port U message {
        out sendrequest, datarequest;
        in sendconfirm, disindication, monitorComplete,
    }

// Definition of the main test component
    type component Tester {
        port U portU; port L portL;
        // Context variable definitions
        var integer nr;
        var integer counter;
        timer timer1 := 10.0; // default timeout 10 sec
        //...
        timer maxDelay := 60.0; } // timer for respons waiting

// Transition implementation definitions
// Transition t1
    function t1() runs on Tester {
        !U:sendrequest
        portU.send (sendrequest_l);
        // ?L:cr
        maxDelay.start;
        alt [[]] portL.receive (cr_1) {maxDelay.stop}
        [any] port.receive {setverdict (fail); stop }
        [any] maxDelay.timeout {setverdict (fail); stop }
    }

testcase TC1() runs on Tester {
// test sequence
    t1(); t7(); t1(); t2(); t3(); t4(); t6(); t4(); t9(); t10();
    t9(); t2(); t4(); t9(); t11(); t16(); t1(); t2(); t3(); t5();
    t4(); t8(); t7(); t15(); t13(); t14();
    setverdict (pass); }

// Module control part
    control {execute (TC1());}
}

Figure 6.1: Fragments of the generated test code in TTCN-3
References


