Generation of Optimized Testsuites for UML Statecharts with Time

Tilo Mücke and Michaela Huhn*

Technical University of Braunschweig, 38106 Braunschweig , Germany,
{tmuecke,huhn}@ips.cs.tu-bs.de,
www.cs.tu-bs.de/ips

Abstract. We present an approach to automatically generate time-optimized coverage-based testsuites from a subclass of deterministic statecharts with real-time constraints. The algorithms are implemented as a plugin for a standard UML tool (Poseidon for UML). The statecharts are extended to accomplish common and new coverage criteria inspired by the experience of test experts and translated into timed automata. The model checker UPPAAL then searches a trace with the fastest diagnostic trace option which provides the basis for the testsuite.

1 Introduction

Model based software development is applied successfully in many application domains. In the embedded domain, the model based approach is well accepted since several years. Tool supported graphical modelling languages are a great help to master the complexity of modern embedded applications that results from safety requirements, the distribution of the components or real-time constraints. Statecharts introduced by Harel [1] are widely used in state based modelling and in particular the UML variant of statecharts is supported by various tools.

In practise, testing is the major technique for software validation and an important expense and time factor in the software development process. For embedded software, testing has become the predominant effort in the development since enhanced safety and reliability requirements have to be guaranteed if the software is employed in hundreds of technical everyday products or highly sensitive systems.

A straightforwared idea for automated test design is so-called model based testing, i.e. to generate tests from (semi-)formal state based design models. In the area of communication systems FSM based testing has been exercised since several decades [2,3,4] and was transferred to statecharts e.g. in [5]. Since the behaviour of a statechart is infinite in general, exhaustive testing is impossible. Thus it is common practise to create a testsuite, i.e. a finite set of tests that cover the system with respect to certain criteria. In software testing coverage criteria related to control flow like state or transition coverage and criteria related to the

* This research was supported in part by a grant of the DFG (Deutsche Forschungsgemeinschaft) - SFB562.

data flow are well established [6,7]. Alternatively, functional queries generalizing the experience of test experts are used to create a small but expressive testsuite [8]. Recently, many authors [9,10,11,12,13] employ a model checker or other efficient search algorithms [14] to cope with the state explosion problem which is dominant in automated testcase generation. The procedure of this approach is depicted in Figure 1.

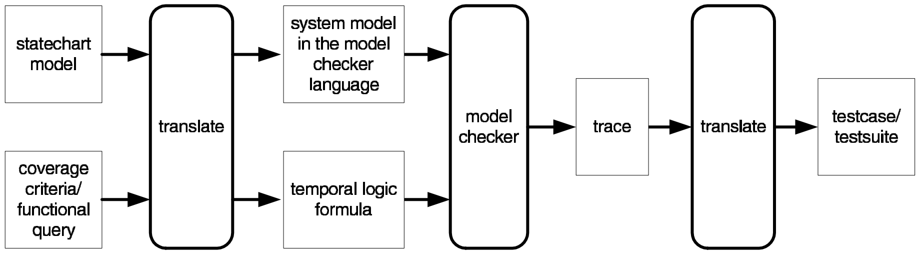


Fig. 1. Using a model checker for testcase generation

In this paper we use a model checker for testsuite generation from statecharts, too. Our approach concentrates on testcases to validate the real-time behaviour of statecharts because in the real-time domain the correct timing of operations is as important as pure functionality. We use the UPPAAL model checker [15] that is specialized for the verification of real-time systems.

As [10] we start with statecharts as a standard modelling notation which is automatically transformed into a formal model suitable as input to a model checker. Depending on the coverage criterion for which a testsuite shall be generated the model is prepared by introducing specific variables. The coverage criterion or functional query is translated into a temporal logic formula, e.g. for state coverage into a query for a path on which each state is visited which is indicated by the introduced variables. Here we follow the work of [10,16] and [12, 9] (for real-time systems) which we have extended by a new coverage criterion called *boundary coverage*. Then the model checker searches the state space of the model which results in a trace that is retranslated to be interpreted as a testcase.

For testcase generation tool support is mandatory ([17] gives an overview). [10,16,18] start with a formal system model ready for a model checker like Spin or SMV, [9,12] use timed automata as the system model. [3] (ObjectGeode), [5] (Rational Rose), and [8] (AutoFokus) have implemented tools as additional modules for various CASE tools. This corresponds to our approach. So the user can conveniently move from modelling to testcase generation and the formal methods behind are mainly hidden.

Even with the excellent algorithms implemented in model checkers, state explosion still is the major problem in testcase generation. [16] shows that the generation of the shortest testsuite is NP-complete and suggests the use of a greedy-algorithm to choose an adequate subset from the generated testcases.

Since version 3.3, UPPAAL contains the feature of emitting the fastest diagnostic trace. This feature provides the generation of the testsuite with the shortest test-execution-time as is shown in [9]. We use this feature to obtain a time-optimal testsuite. However, we show that the search for the fastest trace heavily suffers from the state explosion. Therefore we discuss several heuristics to palliate the storage consumption of the model checker for the price of a time-optimized but possibly non-optimal testsuite.

The rest of the paper is organized as follows: In Section 2 we briefly describe how to transform a family of deterministic statecharts into a system of UPPAAL timed automata. In Section 3 we explain how the statecharts are extended to accomplish the coverage criteria. Section 4 is concerned with our tool *TestGen* and Section 5 with an example. In Section 6 we evaluate the approach and discuss heuristic improvements. Section 7 presents concluding remarks.

2 Translation from Statecharts to Timed Automata

Our approach to generate testcases for statecharts is based on the transformation of UML statecharts to UPPAAL timed automata introduced in [19]. The transformation is restricted to a subclass of UML statecharts (condition 1-3). We put a fourth constraint on the set of suitable statecharts which is specific for testcase generation.

1. Concurrency is restricted to the top level (object level), i.e. within the statecharts AND-states must not be used. This is adequate in our system model, where statecharts model the behaviour of a family of objects without intra-object concurrency.
2. In guards and actions only expressions may be used that have a one-to-one correspondence in the UPPAAL language.
3. So far composite transitions, history connectors, entry-, exit-actions and do activities are not supported by the transformation tool, but these elements can be handled by an extension.
4. Statecharts have to be deterministic in the sense that in the semantics at most one transition is enabled at each moment. Therefore we require that for each two transitions with the same source state the enabling conditions (triggers, guards, timing constraints) are never satisfied both.

In addition to the UML standard syntax, an *after* construct with two parameters is supported to specify real-time constraints: The parameters give a time interval measured from the moment the state has been entered in which the source state may be left via this timing transition.

2.1 Syntax of the Statechart Model

Statecharts extend finite state machines by the concepts of hierarchy, concurrency and communication via events. A UML statechart model consists of states and transitions.

States can be basic or composite. A composite state has at least one substate, while the basic state has none. This hierarchy might be considered as a tree with composite states as internal nodes and basic states as leaves. The root of the tree is the top level composite state containing a complete statechart. Each composite state contains one initial state defining the default entry point. A composite state may contain at most one final state by which the composite state can be left via a so-called completion transition.

Transitions connect a source state with a target state. A transition is labelled by an expression $e[g]/a$, where:

1. e is an event triggering the transition. It can be a signal event, which has been sent from a concurrent statechart, a time event, triggering the transition in a given time interval after entering of the source state, or a completion event, if there is no explicit trigger.
2. g is a guard which has to be evaluated to true, to allow the transition to fire. For a straightforwarded translation to UPPAAL we allow only conjunctions of simple expressions[1].
3. a is a list of actions, which are executed when the transition fires. Actions can be assignments which conform to UPPAAL as above or send actions.

2.2 Statecharts Semantics

Our semantics for statecharts conforms to the UML standard with an extension to handle a family of concurrent statecharts on the object (top) level.

At the beginning, root and all recursively reachable initial substates are marked *active*. All variables are initialized.

A transition is *enabled* if its source is an active state and its trigger is the first in the event queue, in case of a signal or completion event, or if the correct time is reached, in case of a time event, and if its guard evaluates to true.

If several transitions are enabled at the same time, the one with its source state lower in the state hierarchy, has a higher priority. Since the statecharts are deterministic as explained above, with the priority scheme exactly one transition is left.

If a transition fires, the source state and all recursively reachable active substates are left and the target state and all recursively reachable initial substates are marked *active*. The actions are performed in the order of their occurrence. If the action is an assignment, the variable on the left hand is assigned its new value according to the evaluation of the right hand side. If the action is a send action, the send event is enqueued in the event queue of the receiver object.

Signal and completion events are stored in event queues, but completion events are always inserted at the beginning of the queue. If an event causes a transition to fire or is not able to trigger any transition, it is dequeued.

All variables are global.

[1] We only allow real-time clocks and integer variables. Timing contraints are restricted to expressions $c_1 \approx x \approx c_2$ and $c_3 \approx x - y \approx c_4$ where c_i is a non-negative constant or ∞, x, y are real-time clocks, and $\approx \in \{=, \leq, \geq\}$, see [15] for details.

For a basic state, a completion event is generated, whenever the state is entered. For a composite state, a completion event is generated, if its final state is entered.

A family of statecharts is executed in parallel and communicates via events which are addressed to a specified receiver object. Time elapses only, if all event queues are empty. If more than one statechart has a non-empty event queue, the next one to become active is selected non-deterministically. It consumes and processes an event without time delay. This will be repeated without time consumption until all queues are empty. When the system is stable, time elapses and the next time event starts a new sequence of steps.

2.3 Transformation into Timed Automata

A family of UML statecharts is transformed into UPPAAL timed automata in three steps: First, each statechart is flattened, meaning the hierarchical structure is removed because timed automata lack hierarchy. Next, the family of flat statecharts is transformed to a family of timed automata. In a last step, control automata are added to enforce the UPPAAL model to behave consistently to the statechart semantics. The transformation is described formally and in detail in [19].

After flattening the statecharts, the states are translated into locations and transitions into switches. Since we restricted the transition labels accordingly, the translation of transitions triggered by signal or completion events is straightforward.

Next, the after events $\mathit{after}(min, max)$ are translated. For each state s with a leaving transition triggered by an after event, a clock c_s is introduced. Whenever s is entered, the clock c_s is reset. The after event is replaced by two guards on one transition $c_s > min, c_s < max$. To prevent the timed automaton from staying in the state s forever, an invariant $c_s <$ MAX, with MAX being the maximum of all after transitions leaving s, is added to s.

In UPPAAL timed automata, two transitions may synchronize, but there is no way of asynchronous communication. Thus, event queues are modelled explicitly in the timed automata model.

In statecharts events that may not trigger any transition are removed from the head of the queue, thus in the timed automata model, message consuming self loops are added to all states where for some variable configuration and signal or completion event there is no corresponding transition leaving the state.

In UPPAAL timed automata, enabled transitions need not proceed, hence a control automaton, which enforces a timed automaton with a non-empty event queue to fire, is added.

3 Test Generation

To generate test cases we follow a coverage criteria based approach. Control flow can be covered by state, transition, condition, and boundary coverage. Data flow coverage is possible, too.

For some coverage criteria it is sufficient to generate queries to obtain test cases. To find a trace covering a state s, the query $E <> s$ (there exists a path reaching the state s) might be used. But for other coverage criteria, even a CTL^*-formula would not be sufficient [16].

Thus we decided to add Boolean coverage variables c_i to the model. These variables are set to true, whenever a certain coverage is achieved, e.g. a certain state is entered in case of state coverage. To find a trace providing the desired coverage a query $E <> \bigwedge_i c_i$ is used. The augmentation of coverage variables enlarges the statespace as mentioned in Section 6.

State Coverage requires a set of testcases (traces), so that every state is visited at least once. This is done by adding a new Boolean variable for each basic state and adding an assignment to every transition setting the new variable appropriately to the target basic state of the transition to *true*. The initial set of states is visited before any transition fires, therefore the coverage of these states does not need to be verified.

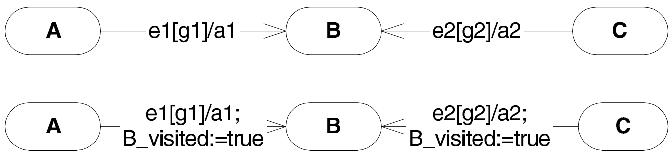


Fig. 2. A first statechart fragment and its state coverage extension

The results of this transformation can be observed in Figure 2.

It is reasonable for all coverage criteria that a coverage must only be achieved if possible. That means, if a state is unreachable, it does not have to be reached to accomplish state coverage.

The name spaces for the variables of the genuine statechart and the added coverage variables have to be disjoint. If necessary, variables have to be renamed before transformation.

Transition Coverage demands a testsuite in which every transition fires at least once. To do this, a Boolean coverage variable for each transition has to be added to the model. Next the actions for each transition are extended, so that the appropriate variable is set to *true*.

The statechart fragment from Figure 2 is therefore transformed to Figure 3. As can be seen, transition coverage is stronger than state coverage.

Condition Coverage can be achieved, evaluating the guard of every transition of the statechart at least once to *true* and once to *false*. To achieve multiple condition coverage [6], the expressions within every guard of each transition of

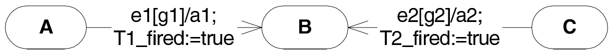


Fig. 3. Transition coverage extension for the first statechart fragment

the statechart will have to be evaluated at least once to *true* and at least once to *false*. Multiple condition coverage is stronger than condition coverage.

Because in our model the atomic guards are combined using the and-operation, it is only possible to prevent (block) the transition from firing by evaluating one atomic guard to *false*. This is why we use an alternative of condition coverage which is in combination with transition coverage stronger than normal condition coverage, but weaker than multiple condition coverage.

For every state, all outgoing transitions are considered and loop-transitions are added for each combination of one blocking atomic guard from every transition. For this coverage a new Boolean variable for every atomic guard is added.

Condition coverage should be used on the flattened statechart. This is no problem, because the statechart is flattened in the first step of the transformation to timed automata. Using our alternative of condition coverage on the statechart before flattening could increase the execution time of test generation and test execution without providing a better coverage.

As an example, the transformation of a statechart fragment with two transitions each with two atomic guards is shown in Figure 4.

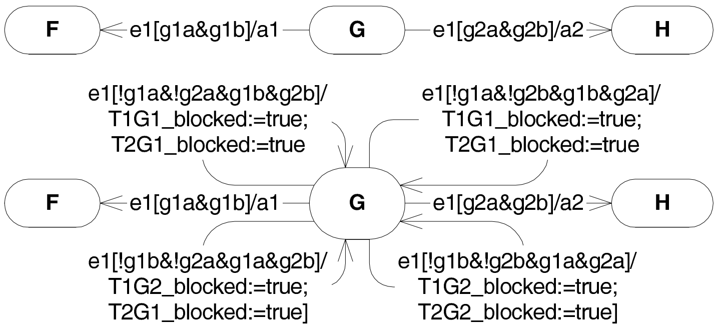


Fig. 4. Condition coverage extension for the second statechart fragment

Boundary Coverage requires that every guard with a relational operator enables its transition to fire at least once with the closest operands possible. Boundary coverage is used to test the limits of guards explicitly, because these are common failure sources.

Boundary coverage is a new development and is achieved by splitting each transition with the relation <, <=, > or >= in its guard in two transitions:

- $a < b$ is splitted in $a = b - 1$ and $a < b - 1$ (only for $a, b \in \mathbb{Z}$)
- $a <= b$ is splitted in $a = b$ and $a < b$

- $a > b$ is splitted in $a = b + 1$ and $a > b + 1$ (only for $a, b \in \mathbb{Z}$)
- $a >= b$ is splitted in $a = b$ and $a > b$

The remaining guards, the triggering event, and the actions are copied to both transitions. The first transition gets an extra action, setting the corresponding Boolean boundary coverage variable to true.

The effect of this transformation can be seen in Figure 5.

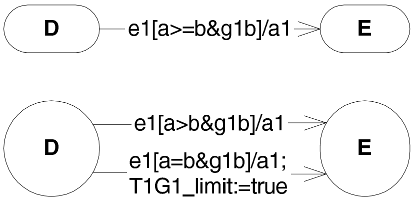


Fig. 5. Boundary coverage extension for the third statechart fragment

Data Flow Coverage demands that every path from an assignment of a variable to the usage of this variable without reassignment is used at least once.

To achieve this kind of coverage, for every variable another variable memorizes where this variable has been set previously. Whenever a variable gets used, a field of a matrix over the definitions and the usages of one variable is set to *true*. The usages are partitioned in predicate uses (p-use) and uses in all other expressions (c-use).

This transformation can be understood more easily looking at the statechart fragment and its transformation in Figure 6.

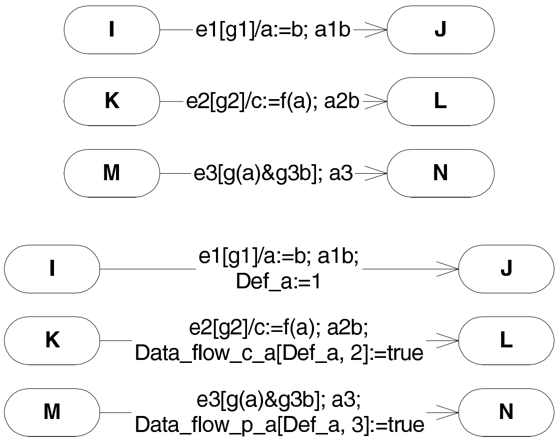


Fig. 6. Data flow coverage extension for the fourth statechart fragment

3.1 Reset Automaton and Satisfiability

Sometimes it is not possible to find a path for some coverage variable because e.g. a state cannot be reached or a transition can never fire. Coverage variables belonging to unreachable features are found, using a query $A[]c == \mathit{false}$ for $c \in C$, and are eliminated. A warning for the designer of the statechart is generated, too.

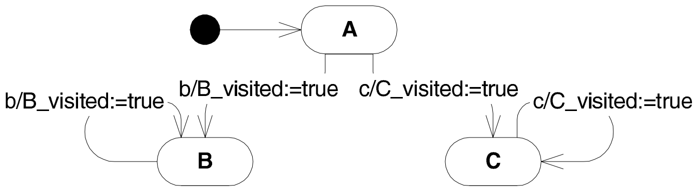


Fig. 7. An example statechart transformed for state coverage

Even if all of the coverage variables can be set to true, in general it is not the case, that a combination of all coverage variables will produce a trace, too. Figure 7 shows a statechart, where the states B and C can both be reached, but not in one trace.

To overcome this situation, a possibility to reset the whole system has to be added to the system. Because a reset of the system takes time in reality, it should take time in the model, too.

Therefore we introduce a synchronous *reset* event triggering transitions leading from every state of a statechart to its new *reset* state. After a given *reset-time* another transition, whose target is the *root* state, fires. This transitions resets all variables except the coverage variables. Because our statechart model does not support synchronous events, the reset-automaton is added after the transformation of the statechart to the UPPAAL-model.

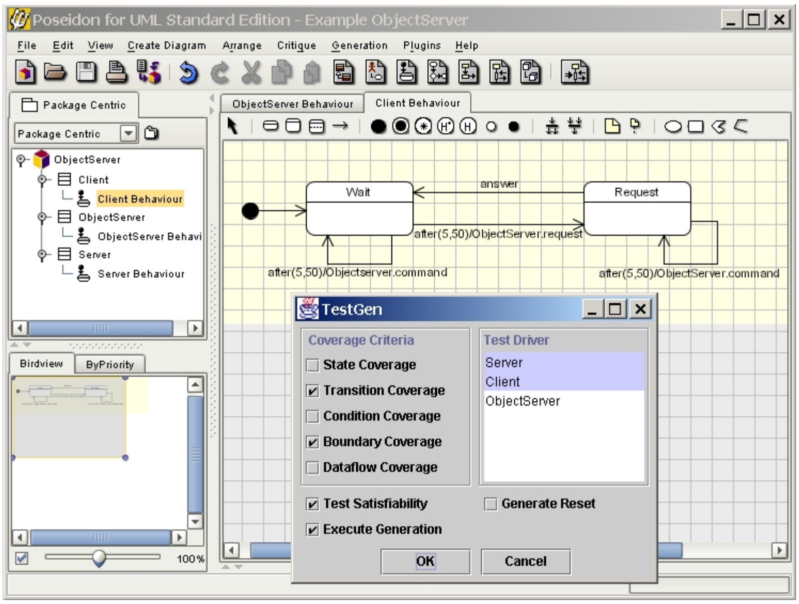
3.2 Test Driver

Every statechart that is part of the system, can be transformed into UPPAAL-automata, but there are two disadvantages:

- A system consisting of several concurrent statecharts causes an explosion in the statespace for the model checker.
- The components are only tested to work correctly in this system but reused components need to be tested again in the new system.

Therefore, the user shall divide the components in two groups. The first group contains the components which shall be tested by the generated testsuite. The second group contains the components for which only test drivers have to be generated.

The test drivers must guarantee the following properties:


Poseidon for UML Standard Edition - Example ObjectServer
File Edit View Create Diagram Arrange Critique Generation Plugins Help
Package Centric
ObjectServer
Client
Client Behaviour
ObjectServer
ObjectServer Behavi
Server
Server Behaviour
Birdview
ByPriority
100%
ObjectServer Behaviour
Client Behaviour
Wait
Request
answer
after(5,50)/ObjectServer.request
after(5,50)/Objectserver.command
after(5,50)/ObjectServer.command
OK
Cancel

TestGen

Coverage Criteria
- [ ] State Coverage
- [✓] Transition Coverage
- [ ] Condition Coverage
- [✓] Boundary Coverage
- [ ] Dataflow Coverage

Test Driver
- Server
- Client
- ObjectServer

- [✓] Test Satisfiability
- [ ] Generate Reset
- [✓] Execute Generation

Fig. 8. Screenshot of TestGen

– Replacing a component by its test driver does not inhibit any behaviour of the remainder of the system.
– The test driver needs less state space than the original component.

To accomplish these properties, the test driver should to be able to:

– send any event, the component has been able to send before,
– change every shared variable to any possible value, if the component has been able to modify this variable before,
– receive every event,
– and wait any amount of time.

This can be done with a simple UPPAAL-automaton with only one state and several self loops, changing the shared variables and sending events.

4 The *TestGen*-Plugin

Our tool "TestGen" (see the screenshot in Figure 8), implementing the algorithms described in Section 2 and 3 in JAVA, has been realized as a plugin for the UML tool *Poseidon for UML*. A testsuite for a family of suitable statecharts modelled in *Poseidon for UML* can be generated as follows:

– Activate the testsuite generation by the plugin button (rightmost button in the menu).

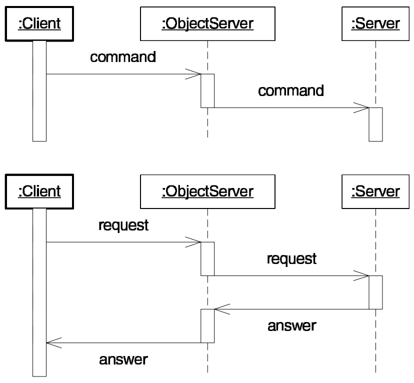


Fig. 9. The two modi of communication provided by the object server

- In a Pop-up menu the user may select:
  - which coverage criteria shall be used
  - which components shall build the test drivers
  - if the satisfiability for each coverage variable shall be tested
  - if a reset automaton shall be included
  - if the generation of the testsuite shall be started after the model transformation
- Then pressing the "Ok"-button starts the testsuite generation

In the moment, the output is a sequence of testcases in a textual representation. We are working on a graphical display of testcases in terms of sequence diagrams.

5 Example: Object Server

As a case study we consider a lightweight real-time middleware called *object server* which was developed within the Sonderforschungsbereich (SFB) 562[2]. The object server [20] serves as a middleware for highly dynamic processes of robot controls and builds a gateway from the external sensors communicating via a high speed industrial communication protocol (IAP) based on the IEEE 1394 standard (firewire) to the main processes of the robot control.

The object server supports two modi of data exchange (see Figure 9):

- command mode: The client sends an asynchronous command to the server which starts executing it.
- request mode: The client sends a request to the server which calculates an answer and sends it back to the client.

Figure 10 shows a simplified version of the object server. The model contains only one server and client and does not include the data possibly associated with

[2] The SFB 562 is promoted by the DFG (Deutsche Forschungsgemeinschaft).

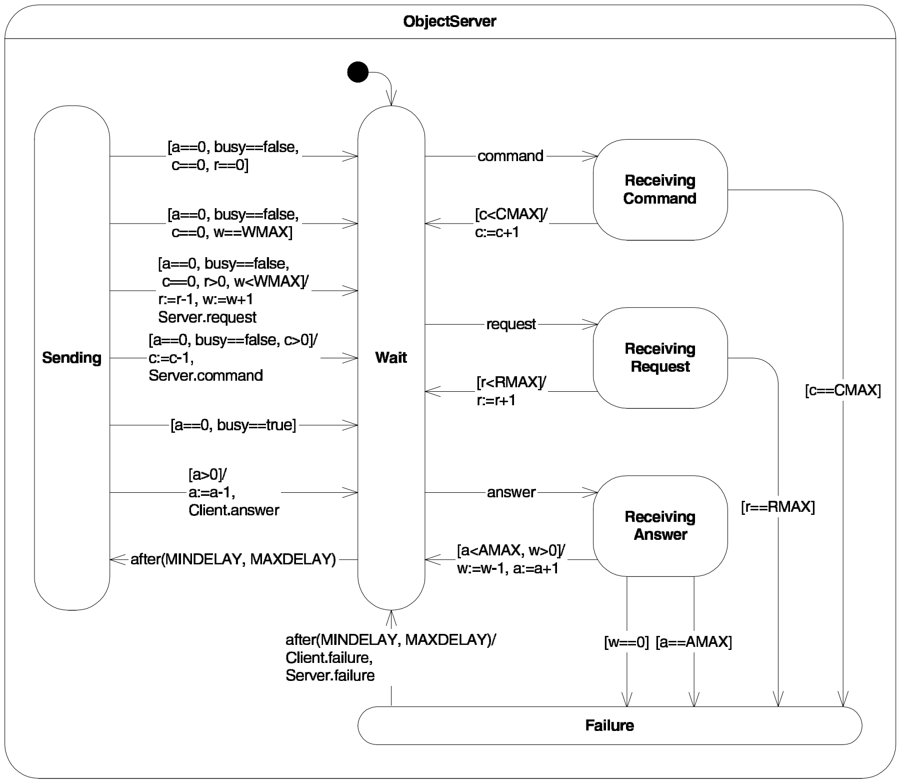

ObjectServer
Sending
Wait
Receiving Command
Receiving Request
Receiving Answer
Failure
[a==0, busy==false, c==0, r==0]
[a==0, busy==false, c==0, w==WMAX]
[a==0, busy==false, c==0, r>0, w<WMAX]/ r:=r-1, w:=w+1 Server.request
[a==0, busy==false, c>0]/ c:=c-1, Server.command
[a==0, busy==true]
[a>0]/ a:=a-1, Client.answer
after(MINDELAY, MAXDELAY)
command
[c<CMAX]/ c:=c+1
request
[r<RMAX]/ r:=r+1
answer
[a<AMAX, w>0]/ w:=w-1, a:=a+1
[c==CMAX]
[r==RMAX]
[w==0]
[a==AMAX]
after(MINDELAY, MAXDELAY)/ Client.failure, Server.failure

Fig. 10. Statechart of the object server

the messages, but it can store incoming messages in several queues and delivers them, when the server is not busy with incoming messages. An error occurs in case of queue overflows and in case of a mismatch between requests and answers. Then the server and the client are informed.

If a command, a request, or an answer is received, it is stored in the corresponding queue. The fill status of the queues is memorized in the variables c, r and a. If the appropriate queue is already full or an answer is received which is not expected, the object server sends a failure message to both, the server and the client. If no message arrives for a given period of time (MINDELAY, MAXDELAY), the object server starts to deliver messages from the queues. Answers have the highest priority of delivery. Commands may only be delivered if the queue for the answers is empty and the server is not busy. Requests may only be delivered if the other queues are empty and the server is not busy.

Generating a testsuite for this statechart using transition and boundary coverage results in the testcase which is depicted in Figure 11. Modifying the model requires only a single run of the testcase generator to automatically generate a new testsuite.

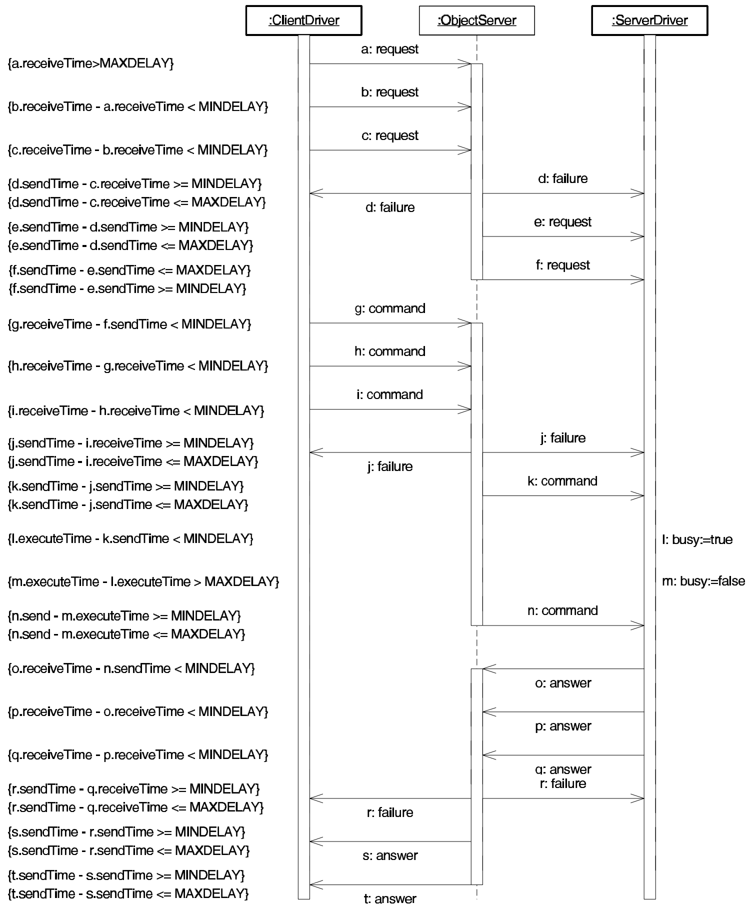

:ClientDriver
:ObjectServer
:ServerDriver
a: request
{a.receiveTime>MAXDELAY}
b: request
{b.receiveTime - a.receiveTime < MINDELAY}
c: request
{c.receiveTime - b.receiveTime < MINDELAY}
d: failure
{d.sendTime - c.receiveTime >= MINDELAY}
{d.sendTime - c.receiveTime <= MAXDELAY}
d: failure
e: request
{e.sendTime - d.sendTime >= MINDELAY}
{e.sendTime - d.sendTime <= MAXDELAY}
f: request
{f.sendTime - e.sendTime <= MAXDELAY}
{f.sendTime - e.sendTime >= MINDELAY}
g: command
{g.receiveTime - f.sendTime < MINDELAY}
h: command
{h.receiveTime - g.receiveTime < MINDELAY}
i: command
{i.receiveTime - h.receiveTime < MINDELAY}
j: failure
{j.sendTime - i.receiveTime >= MINDELAY}
{j.sendTime - i.receiveTime <= MAXDELAY}
j: failure
k: command
{k.sendTime - j.sendTime >= MINDELAY}
{k.sendTime - j.sendTime <= MAXDELAY}
{l.executeTime - k.sendTime < MINDELAY}
l: busy:=true
{m.executeTime - l.executeTime > MAXDELAY}
m: busy:=false
n: command
{n.send - m.executeTime >= MINDELAY}
{n.send - m.executeTime <= MAXDELAY}
{o.receiveTime - n.sendTime < MINDELAY}
o: answer
{p.receiveTime - o.receiveTime < MINDELAY}
p: answer
{q.receiveTime - p.receiveTime < MINDELAY}
q: answer
r: failure
{r.sendTime - q.receiveTime >= MINDELAY}
{r.sendTime - q.receiveTime <= MAXDELAY}
r: failure
{s.sendTime - r.sendTime >= MINDELAY}
{s.sendTime - r.sendTime <= MAXDELAY}
s: answer
{t.sendTime - s.sendTime >= MINDELAY}
{t.sendTime - s.sendTime <= MAXDELAY}
t: answer

Fig. 11. Testsuite for the object server

6 Problems of the Method

The main problem using our approach is an exponential coherence between the length of a trace and the amount of memory used by UPPAAL for testcase generation, as depicted in Figure 12. This problem has two main causes:

- The state space is enlarged by the boolean coverage variables.
- A long trace is searched, because the whole testsuite is calculated by one single run of the model checker.

Both problems can be overcome with a simple variation of the algorithm.

1. In the first step, for some coverage variable not reached so far a path is calculated.

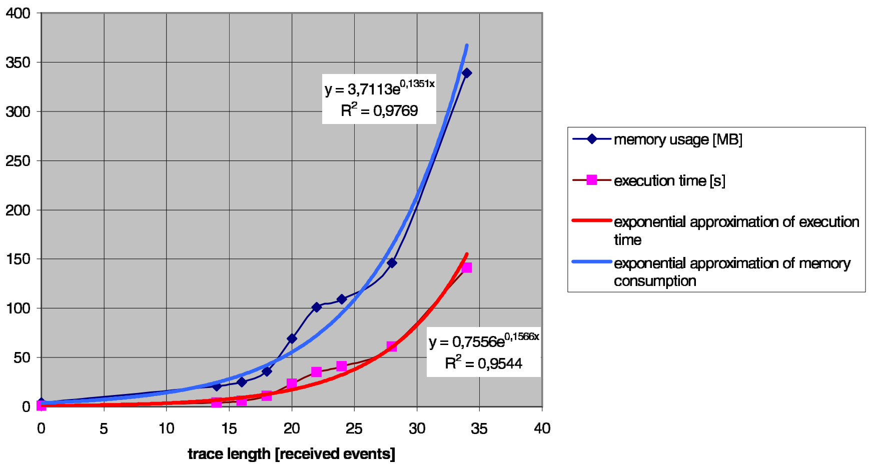


Fig. 12. Complexity trace-length dependency

2. Next, for the calculated path the coverage variables are determined which are reached additionally to the searched one.
3. If there are still reachable coverage variables which are not reached so far then we proceed to step 1 with the end configuration of the calculated path as a new start configuration.

Doing so, only one coverage variable is needed. The trace length of each run of the model checker is shortened, for the price of starting the model checker several times. The improvement of the algorithm results in lower memory consumption but higher execution time for the model checker.

This kind of search can be improved by a branch and bound algorithm like A^* [21], but has one disadvantage: It is not assured any more that the shortest testcase is generated if the implementation reacts as fast as possible. However, it is obvious that the concept provides shorter testsuites than the greedy algorithm. Heuristics like testcase ranking, to cut off some search paths, can be incorporated in the search algorithm, to improve the execution time of the testcase generation.

7 Conclusion and Future Work

We have presented a method for the automated generation of time-optimized testsuites for UML statecharts. The method is based on a transformation of statecharts to timed automata. For testcase generation the tool adds coverage variables and a reset automaton to the UPPAAL model. The testsuite is derived from a trace generated by UPPAAL in a model checking run with the "fastest trace"-option. Components can be tested stand-alone using test drivers for their environment, so that they can be reused without further testing. In difference to other approaches to testcase generation for statecharts using model checkers like Spin and SMV [18,10] our work aims for the testing of the timing behaviour.

In addition to [9] we offer tool support as a plugin for *Poseidon for UML* and an additional coverage criterion. To reach our goal to develop an environment for automatic conformance testing of a model and its implementation, there are several extensions to be explored:

Variations of the generation algorithm. Next we will implement the improvements mentioned in Section 6 to overcome the problems resulting from the enormous memory consumption. For us it seems to be more promising to optimize the testcase generation procedure for the price of good but non-optimal testsuites than to keep to the time-optimal testsuite, because the memory problem is the major restriction for practical applications. The algorithm would benefit further from the use of an optimized model checker and a modifiable search strategy as recommended in [14].

Syntax checks on statecharts. For the convenience and understanding of the user, we plan to implement a syntax check on statecharts that points out model elements that cannot be processed by TestGen.

Test execution via middleware will allow to run the test suite without adaption of the implementation. The components in the environment of the CUT (component under test) will be replaced by the corresponding test drivers, each of which sends the events noticed in each testcase to the CUT and checks its reactions for compliance with the testcase.

References

1. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8** (1987) 231–274
2. Bochmann, G., Petrenko, A.: Protocol testing: Review of methods and relevance for software testing. In: Proc. International Symposium on Software Testing and Analysis. (1994) 109–124
3. Kerbra, A., Jéron, T., Groz, R.: Automated test generation from SDL specifications. In: SDL Forum. (1999) 135–152
4. Rapps, S., Weyuker, E.: Selecting software test data using data flow information. In: IEEE TSE. Volume 11. (1985) 367–375
5. Offutt, J., Abdurazik, A.: Generating tests from UML specifications. In: UML'99. (1999) 416–429
6. Peled, D.: 9. Software Testing. In: Software Reliability Methods. Springer-Verlag (2001)
7. Friedman, G., Hartman, A., Nagin, K., Shiran, T.: Projected state machine coverage for software testing. In: ACM SIGSOFT European Software Engineering Conference and International Symposium on Foundations of Software Engineering. (2002) 134–143
8. Pretschner, A., Lötzebeyer, H.: Model based testing with constraint logic programming. In: Workshop on Automated Program Analysis, Testing and Verification (WAPATV). (2001) 1–9

9. Hessel, A., Larsen, K., Nielsen, B., Pettersson, P., Skou, A.: Time-optimal real-time test case generation using UPPAAL. In: Workshop on Formal Approaches to Testing of Software (FATES). (2003)
10. Hong, H., Lee, I., Sokolsky, O., Cha, S.: Automatic test generation from statecharts using model checking. In: Workshop on Formal Approaches to Testing of Software (FATES). (2001) 15–30
11. Rayadurgan, S., Heimdahl, M.: Coverage based test-case generation using model checkers. In: Intl. Conf. and Workshop on the Engineering of Computer Based Systems. (2001) 83–93
12. Nielsen, B., Skou, A.: Automated test generation from timed automata. In: Tools and Algorithms for the Construction and Analysis of Systems. (2001) 343–357
13. Jéron, T., Morel, P.: Test generation derived from model-checking. In: International Conference on Computer Aided Verification. Volume 1633. (1999)
14. Pretschner, A.: Classical search strategies for test case generation with constraint logic programming. In: Workshop on Formal Approaches to Testing of Software (FATES). (2001) 47–60
15. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer **1** (1997) 134–152
16. Hong, H., Lee, I., Sokolsky, O., Ural, H.: A temporal logic based theory of test coverage and generation. In: Tools and Algorithms for the Construction and Analysis of Systems. (2002)
17. Goga, N.: Comparing TorX, autolink, TGV and UIO test algorithms. Lecture Notes in Computer Science (2001)
18. Gragantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specification. In: ACM SIGSOFT European Software Engineering Conference and International Symposium on Foundations of Software Engineering. (1999) 146–162
19. Diethers, K., Goltz, U., Huhn, M.: Model checking UML statecharts with time. In: UML 2002, Workshop on Critical Systems Development with UML. (2002)
20. Diethers, K., Kohn, N., Finkemeyer, B.: Middleware zur Realisierung offener Steuerungssoftware für hochdynamische Prozesse. it - Information Technology (2003)
21. Nilsson, N.: Principles of Artificial Intelligence. Springer Verlag (1982)