Model-based Automatic Test Generation for Event-Driven Embedded Systems using Model Checkers

Zoltán Micskei, István Majzik
Budapest University of Technology and Economics
Dept. of Measurement and Information Systems
Magyar Tudósok krt. 2., H-1117 Budapest, Hungary
micskeiz@mit.bme.hu, majzik@mit.bme.hu

Abstract
Testing is an essential, but time and resource consuming activity in the software development process. In the case of model-based development, among other subtasks test construction and test execution can be partially automated. Our paper describes the implementation of a test generator framework that uses an external model checker to construct test sequences. The possible configurations of the model checker are examined by measuring the efficiency of test construction in the case of different statechart models of event-driven embedded systems. The generated test cases are transformed and executed on common testing frameworks (JUnit, Rational Robot) and the effectiveness of tests are measured using code coverage metrics.

1. Introduction
Testing is a time and resource consuming activity in the software development process. Typically more than 30% of efforts should be reserved for testing activities. Generating a short, but effective test suite usually needs a lot of manual work and expert knowledge. A testing engineer’s traditional tasks are writing test cases (input-output pairs) for the important functions, grouping them in test sequences and test suites, then executing the tests and finally analyzing the results. Test suites should satisfy various test coverage criteria often prescribed by standards (e.g. all states of the program have to be visited, all paths have to be executed). A criterion defines a set of test requirements (e.g. visiting given states, executing specific state transitions). The coverage of a test suite is the ratio of satisfied requirements and all requirements. Various specification and code-based criteria are presented in [1].

In a model-based development process test construction and test execution can be partially automated. In a (semi-)formal model all interfaces and possible input events are gathered which forms a suitable basis to start the test construction. This way model-based testing can be a solution to several problems of test generation. The following tasks can be automated:
- **Test oracle.** The model is used to derive the required output for a given test input. A quite high coverage can be achieved in a short time by randomly generating test inputs and deriving the outputs using a test oracle.
- **Estimating the coverage.** The coverage of a test suite can be estimated with the help of a system model [2].
- **Conformance testing.** Running the implementation and the model simultaneously helps to determine whether the implementation conforms to the requirements.
- **Test generation.** Complete test suites can be constructed by using the model and the test criteria defined on the specification. Typically, the state space of the model is searched for sequences of inputs (and outputs) that satisfy the test requirements. This task can be performed in several ways, e.g. implementing a specific search algorithm (like in [3]), using a Constraint Logic Programming solver, or using a model checker [4].
In our paper we present a testing framework that uses external model checker tools to construct the test sequences. The process used for testing is given in Figure 1. This framework is designed for embedded event-driven systems, where the functionality can be described as a sequence of reactions to incoming events from the environment. UML Statechart is a very popular formalism to capture the behavior of such systems. In our framework, we define test requirements on the basis of the statechart model then configure the model checker tool to find a test sequence (test inputs and required test outputs) for each requirement. The generated tests are then transformed to executable, concrete tests, and with the help of a test execution environment these tests are executed on the implementation of the model.

![Figure 1. Testing process in our framework](image)

There are very few publications in the literature that report on the efficiency of using model checkers for test construction. The configuration of the model checker in this case, namely the settings required for constructing short (and possibly minimal) test sequences, differs from the usual needs of the classical model checking problem (i.e. exhaustive verification of the full state space). In our paper we examine the possible settings of the model checker Spin by measuring the efficiency of test construction in the case of different real-life statechart models, and introduce an optimized setting for test generation. At the end of the testing process the code coverage of the automatically generated tests is measured, hence they can be compared to manually created tests.

2. Model checkers and test generation

In safety-critical systems it is necessary not only to test but also to prove that the system works correctly. This activity is supported by model checker tools. They examine the state space of the input model to check the truth of temporal logic expressions that apply temporal operators to express state reachability properties and required temporal ordering of states. The greatest challenge of these tools is the state space explosion as practical models of concurrent systems may have a huge number of states. The limit of state-of-the-art model checkers grows above $10^{20}$ states.

In our experiments we applied the following model checkers:

- **Spin** [5] accepts models of concurrent processes given in Promela, its specific input language that supports global variables and communication channels. The requirements can be formalized by reference automata or by LTL (Linear Temporal Logic) formulae. Partial order reduction and bit state hashing are applied to handle the state space.

- **SMV** is a symbolic model checker that accepts the description of finite state machines. (Its input language has constructs to express state transitions.) Requirements can be formalized by CTL (Computational Tree Logic, a branching time logic) formulae. Internally it uses Binary Decision Diagrams to store and manipulate the state space.

- **Uppaal** [12] is a model checker for real-time systems. Its input model can be given by a set of timed automata including global variables and conditions that refer to clock variables. Requirements are expressed in a CTL-like language.
Model checkers were proposed to be used as test generation tools in various publications. [4] used SMV to generate tests for Statemate statecharts, but as far as we know no tool was developed to automate the process. The AGEDIS project aims at the automation of testing. The developed tool generates tests from specially annotated UML diagrams, and the abstract test suites can then be mapped to concrete ones executed on the implementation [6]. In [7] mutation analysis was used to create tests that can detect different programming errors. The method presented in [8] is similar to [4], but here Spin is used and the input model is an abstract state machine. A prototype of a test generator tool (ATGT) is also reported. There are several other academic and commercial tools available as summarized in [9].

In our experiments we apply the following test generation method:

1. The system model in the form of annotated UML statechart is transformed into the input language of the model checker tool.
2. Each test requirement of a given coverage criterion (e.g. reachability of a given state by a test sequence) is formulated as a temporal logic expression.
3. For each expression the negation of the formula is verified by the model checker. If there is an execution path in the model that does not satisfy the negated formula (e.g. it turns out that a given state can be reached by an execution path) then it is presented by the model checker as a counter-example. This path becomes a test sequence that satisfies the original test requirement.
4. The inputs and outputs that form the executable test sequence are extracted from the counter-example or derived by a corresponding guided simulation of the model.

The necessary model transformation and the configuration of the model checker are detailed in the next sections. The temporal logic based formulation of the set of test requirements is illustrated in the case of the following coverage criteria:

- **All states** coverage: For each state of the statechart a test sequence is generated, that leads the system to that state. The LTL formula-set for this criterion is
  \[
  \neg (F \text{ in}(s)) \text{ } \forall s \in S
  \]  
  where \( F \) is the eventually operator in LTL, \( S \) is the set of states and \( \text{in}(s) \) is a Boolean expression on the state variables of the model that is true when the state \( s \) is active.

- **All transitions** coverage: For each transition in the statechart a test sequence is generated that fires the transition. The LTL formula-set for this criterion is
  \[
  \neg (F \text{ firing}(t)) \text{ } \forall t \in T
  \]  
  where \( T \) is the set of transitions in the model and \( \text{firing}(t) \) is a Boolean expression on transition variables that is true when transition \( t \) is fired.

Other control-oriented test requirements (e.g. coverage of all configurations, transition-pair coverage) can be formulated similarly. Data-oriented criteria (e.g. all definition-use paths) are not covered by our experiments. The temporal logic representation of the corresponding test requirements can be found in [10] and in [4].

3. The test generation framework

Our framework consists of the tools and data files presented in Fig. 2. The box marked with TR is a model transformation from UML statecharts to Promela [11] that uses Extended Hierarchical Automata (EHA) as its intermediate format. The transformation consists of two programs: sc2eha (in Prolog), and eha2promela (in Java). The box marked with TG is the test generator implemented in Java. Its input parameters are the statechart model and the test coverage criteria. It controls the test generation process as follows:
1. The settings of the tools are read from an XML based configuration file.
2. Both sc2eha and eha2promela programs are executed. Here sc2eha builds an EHA representation of the model which is accessed by the test generator to obtain information on the events, states and transitions in the statechart.
3. For each test requirement a file containing an LTL formula is created.
4. The executable pan of the model checker Spin is executed that produces a report of the results and a trail file (i.e. a counter-example used later as the test sequence), if exists.
5. A filtering procedure generates an XML file that contains only the input and output operations from the sequence of atomic actions described in the trail.
6. The temporary files are preserved, in this way if no test is found or Spin runs out of memory then the test generation can be repeated with modified settings.

![Figure 2. Test generator framework](image)

### 4. Configuration of the model checker

Classical model checking aims at the fast exhaustive search of the full state space of the model. In contrary, test generation by model checking aims at the fast and efficient construction of a counter-example by visiting and storing as few states as possible. Not all counter-examples are needed, however the one that is generated should be minimal in length. Due to this difference, the direct (default) use of model checkers typically results in poor efficiency when used for test generation. Fortunately, the well established tools offer several built-in state handling techniques and parameters for tuning that make the optimization of the test construction possible. We performed several experiments to determine the effects of the available options.

The experiments were performed on a 2 GHz PC with 512 MB RAM. The benchmark model was a statechart describing the operation of a mobile phone having 10 states, an event queue and 21 transitions (about 500000 state configurations are visited by Spin).

**Table 1. Execution time of test construction phases**

<table>
<thead>
<tr>
<th></th>
<th>Generate LTL formula</th>
<th>Generate pan</th>
<th>Compile pan</th>
<th>Run pan</th>
<th>Extract tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default case</td>
<td>0,25s</td>
<td>0,70s</td>
<td>44,92s</td>
<td>75,00s</td>
<td>0,71s</td>
</tr>
<tr>
<td>Short tests</td>
<td>0,23s</td>
<td>0,69s</td>
<td>44,37s</td>
<td>553,51s</td>
<td>0,66s</td>
</tr>
</tbody>
</table>

Table 1 shows the execution time of two typical test generation runs. The first case is the default configuration with no test-specific settings, while in the second case Spin was
instructed to find iteratively the shortest counter-example. This is why the execution of the model checker executable `pan` lasted longer (while the time required for formula generation and compilation remained roughly the same). It is important to note that in shorter runs the compilation time and the running time are in the same order of magnitude.

Both the compilation and the execution of `pan` are controlled by several options [5]. The settings interesting from the point of view of test generation are as follows:
- **Breadth first search**: The compilation option `-dBFS` means that instead of the default depth first search (DFS) algorithm breadth first search (BFS) is implemented. It is optimal for test generation since a breadth first search finds the shortest trail. However, the depth of BFS is limited by the available memory. Using the default settings only a three transition long sequence could be generated. BFS is capable of generating all test sequences if the lengths of the event queues are reduced in the model and atomic instruction sequences are used.
- **Limited DFS**: The run-time option `-m` sets a *limit for the DFS*. It is one of the most useful options, since it restricts the depth of the search (i.e. the length of the counter-example). However, if it is set too low then test sequences cannot be found. In this case it is helpful to increment the value until the memory limit is reached.
- **Iterative search for short traces**: `-I` and `-i` options *search for shorter counter-examples* with iterative runs. They could greatly enlarge the execution time as `-i` finds the shortest trail. `-I` is approximate and faster, it performed well in our experiments.
- **Model dependent options**: The usage of `-dNOFAIR` (weak fairness), `-dSAFETY` (cycle detection disabled) parameters did not result in significant improvements.

We found that the following settings are optimal for this model (see in Table 2):
- **DFS search**: All compiling options are used except `-dBFS` and `-dREACH`.
- **No iterative search**: Options `-I` and `-i` are not used since they increase the time required for test construction. Instead of them, DFS is limited by parameter `-m`.
- **Limited DFS**: Option `-m` is set to the minimal value where tests are generated (it can be set in a few probe runs). Changing the value from 500 to 200 resulted in 80% decrease of the execution time in the benchmark model.
- **Hash table size**: Option `-w` controls the size of the hash table used to store the visited states. The value of this switch should be increased in the case of detecting a high number of hash conflicts. In this example the value 24 resulted in a total memory usage of 67 MB.

### Table 2. Execution time and test properties in case of different options.

<table>
<thead>
<tr>
<th>Options</th>
<th>Time required for test generation</th>
<th>Length of the test sequences</th>
<th>Longest test sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>-i</td>
<td>22m 32.46s</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>-dBFS</td>
<td>11m 48.83s</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>-i -m1000</td>
<td>4m 47.23s</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>-l</td>
<td>2m 48.78s</td>
<td>25</td>
<td>6</td>
</tr>
<tr>
<td>Default</td>
<td>2m 04.86s</td>
<td>385</td>
<td>94</td>
</tr>
<tr>
<td>-l -m1000</td>
<td>1m 46.64s</td>
<td>22</td>
<td>4</td>
</tr>
<tr>
<td>-m1000</td>
<td>1m 25.48s</td>
<td>97</td>
<td>16</td>
</tr>
<tr>
<td>-m200 -w24</td>
<td>46.7s</td>
<td>17</td>
<td>3</td>
</tr>
</tbody>
</table>

Using these parameters the generation of test sequences covering all states required 46.7 seconds, while covering all transitions required 4 minutes and 19 seconds. All test sequences were the shortest possible (this was cross-checked manually).

The lessons learnt during these experiments can be summarized as follows:
- The original (default) settings resulted in relatively short execution time but overlong test sequences (the length is 385 instead of 17).
- Iterative search algorithms resulted in short test sequences but long execution time (22 minutes vs. 2 minutes). The same applied to BFS (11 minutes vs. 2 minutes).
- The best results were obtained in the case of limiting the depth of the DFS together with setting a limit to the hash table size.

It turned out that the suitable setting of options could reduce the time required to find the necessary test suite to 37% (comparing to the default case) and to 3.5% (comparing to the pure iterative search). The obvious settings (iterative search) did not perform well.

Additional experiments were performed to compare the efficiency of Spin to SMV. The total execution time of generating test sequences covering all states was 10.98 seconds. SMV outperformed Spin due to the following reasons: In SMV there is no need to compile the verifier and SMV reuses the data structures between different verification runs. However, the Spin model, thanks to the automatic generation, is more generally applicable.

5. A case study

The applicability of the framework was demonstrated in the case of a real-life industrial model. We choose a protocol that synchronizes status and control bits between two computers in a distributed control system in presence of anticipated faults. The model consists of 5 objects with event queues, 31 states and 174 transitions. In the generated Promela code the state vector (which identifies a state) was 216 bytes long, and during the exhaustive verification more than $2 \cdot 10^8$ states were explored. The complete verification would need approximately 40 GB of memory, therefore the application of state compression techniques was necessary.

We examined the effects of the bit-state hashing state compression technique offered by Spin. This technique does not store the entire state vector, only one or two bits per state in the memory. This results in states that are not stored separately but merged with others. Additionally to the state compression technique, the lengths of the event queues of the objects were decreased to the minimum. This kind of reduction modifies the behavior of the model by reducing the length of the sequences that can be produced.

The reduction and compression techniques are applied in a conservative manner. On the one hand, if a test sequence is found in a reduced model (note that typically there are several tests for a given requirement) then it is a valid test sequence executable on the full model. On the other hand, if no test sequence is found for the given requirement then it may happen that a test sequence can be generated on the basis of the full model.

Applying these reductions the test generation was successful, but it required two and a half hours to terminate. Two more adjustments were needed:
- Several test requirements are satisfied by multiple test sequences. Accordingly, the number of the verification runs can be reduced if each test sequence is checked for covering other test requirements.
- Parameter -w (size of the hash table) is more important in bit-state mode. If it is too low, the verification misses too many states, and thus test sequences are lost.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Execution time</th>
<th>Redundant runs skipped</th>
<th>No test found (states)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-m1000 –w31</td>
<td>65m4.32s</td>
<td>65 %</td>
<td>0</td>
</tr>
<tr>
<td>-m1000 –w26</td>
<td>46m2.30s</td>
<td>62 %</td>
<td>2</td>
</tr>
<tr>
<td>-w26</td>
<td>56m47.66s</td>
<td>55 %</td>
<td>4</td>
</tr>
<tr>
<td>-w24</td>
<td>28m3.35s</td>
<td>48 %</td>
<td>8</td>
</tr>
</tbody>
</table>
Table 3 shows that our tool (with proper parameters) can be used to construct test sequences for complex protocols with huge state space. In this case the test suite consists of 31 test sequences if a separate test is generated for each state. The length of a test sequence is typically 11 steps. If the redundant tests are eliminated then 3 long and several short sequences (typically 3-4 steps) are required.

6. Applying tests to implementation

The tests generated by our framework are abstract, i.e. they correspond to the events and actions defined in the model. If we want to use them to test concrete implementations, then the test cases should be mapped to concrete test cases which are executable on the real System Under Test (SUT). The process of this transformation was demonstrated on the mobile example, constructing a Java GUI based implementation and a console-based implementation (this latter was generated automatically). A concrete test case contains the following parts:
- Setup code to establish the necessary state required to start the test.
- Code for sending the events, and checking that expected actions can be observed.
- Teardown code to clean up any resources created by the test.

Our transformation used templates to automate concrete test case generation. One template contained the general structure of the test case file, another included the parameterized code used to send events and check actions. Test cases were generated for the JUnit Java unit testing framework and for the Rational Robot automatic GUI testing tool. Only 10 lines of implementation-specific code had to be inserted into the templates to generate approximately 500 lines of test code.

One of the most important metric to evaluate the efficiency of a test suite is the code coverage it produces. We measured the statement, method and condition coverage the test suite produced for both programs. Figure 3 shows the results.

![Coverage graphs](image)

**Figure 3.** Code coverage results

The abstract test suite (that was characterized by the “all transitions coverage” criterion in the model) covered in the code more than 90% of the statements and methods. The remaining 10% was exception handling code (e.g. handling invalid events), which was not called because test cases contained only valid events. These experiments show that our approach is a cheap and efficient way to generate a base test suite quickly that exercises the majority of the implementation’s code.

7. Testing Real-Time Systems

In the case of real-time systems test sequences shall carry timing information, i.e. the time delay between successive test inputs (and timing parameters of the required outputs). The
execution of such a test sequence shall keep these time delays in order to execute the given path determined by the coverage criteria.

In our framework test sequences can be generated on the basis of UML statechart models extended with timing information (clock variables). In this case the model checker Uppaal [12] is applied that is capable of verifying real-time systems.

Uppaal offers the following functions which are useful for test generation:
- BFS can be used (besides DFS). It provided the shortest tests for relatively small systems.
- It has multi-level state compression and supports bit-state hashing as well.
- Uppaal can reuse existing data structures when multiple properties are checked.
- It can generate the shortest trail without iterative runs (disables reuse).

As a first benchmark, the behavioral model of the mobile phone example was extended with timing information. The transformation from timed UML statecharts to the timed automata of Uppaal faced the following challenges:
- There is no state hierarchy in Uppaal. The transformation should ensure that a sub-automaton is active only when its parent automaton is active.
- There are only binary synchronization channels in Uppaal, which cannot pass values. Event queues were implemented using global arrays and dedicated dispatcher processes.

The results of the first experiments with the mobile phone example were encouraging. The test generation took 24.94 seconds and Uppaal generated the shortest possible test sequences. Uppaal recorded in the trail (counter-example) whenever a delay occurred between the firings of transitions in the model, so the timing information (i.e. the delay between successive test inputs) required executing the test sequence could be extracted from the trail file.

8. Conclusion

We outlined a framework that uses external model checkers to construct test sequences corresponding to state and transition coverage criteria in event-driven and real-time systems. Different configurations of the Spin model checker were examined experimentally to find those settings that are optimal for test generation. We demonstrated that (i) using proper options the time required for test generation can be significantly reduced, (ii) the state compression techniques do not hinder the construction of test sequences in practical models of concurrent systems, and (iii) the generated abstract test produces high coverage on the implementation.

9. References