CO✓ER – A Real-Time Test Case Generation Tool

Anders Hessel[1] and Paul Pettersson[1,2]

[1] Department of Information Technology, Uppsala University, P.O. Box 337, SE-751 05 Uppsala, Sweden. E-mail: {hessel,paupet}@it.uu.se.
[2] Department of Computer Science and Electronics, Mälardalen University, P.O. Box 883, SE-721 23, Västerås, Sweden. E-mail: Paul.Pettersson@mdh.se.

Abstract. CO✓ER is new test-case generation tool for timed systems. It generates test cases from a timed automata model of a system to be tested, and a coverage criteria expressed in an observer language. In this paper, we describe the current architecture of the tool, its input languages, and a case study in which tool has been applied in an industrial setting to test that a WAP gateway conform to its specification.

1 Introduction

CO✓ER is a tool for model-based testing of real-time systems, developed at Uppsala University since 2005. It allows its users to automatically generate test suites from timed automata specifications of real-time and embedded systems. The generated test suites can be compiled into test programs that can be used to automate the execution of a system under test. The main features of the CO✓ER tool are:

- A framework for extending model checkers with coverage-based test case capabilities. I our case, we have used the verifier of the UPPAAL tool [6].
- An observer language [2] that is expressive enough to describe a large set of coverage criteria, including *structural criteria*, such as location or edge coverage, *data-flow criteria*, e.g., definition-use pairs, and *semantic coverage*, such as states or projection of states. Thus, the tool is not limited to a number of predefined criteria, but rather to the expressiveness of the observer language. Together with the CO✓ER tool, we distribute examples of observer specifying a set of popular coverage criteria.
- A query language that is used to specify from which automaton or automata of a model a test suite should be generated. In addition, the level of required coverage (a natural number or the maximum possible coverage) can be specified.
- An efficient test suite generator that generates a test suite with full feasible coverage. CO✓ER uses a novel global algorithm that uses knowledge about the total coverage found in the currently generated state space to guide and prune the remaining exploration [4].
- The CO✓ER tool is compatible with the file format for representing models in the UPPAAL tool. Thus, users can use the graphical editor of UPPAAL to specify system specifications, and generate test cases in CO✓ER.
- A configurable post processor that helps users to format the generated test suite to contain desired information in XML format.

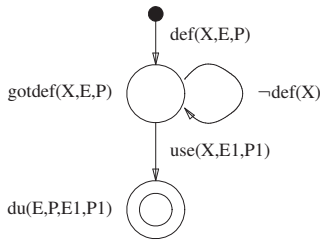

Fig. 1. A graphical representation of a def-use observer.

```
observer du(varid X;) {
  node gotdef(varid, edgeid, procid);
  node du(edgeid, procid, edgeid, procid);

  rule start to gotdef(X,E,P) with def(X,E,P);
  rule gotdef(X,E,P) to gotdef(X,E,P) with no def(X);
  rule gotdef(X,E,P) to du(E,P,E1,P1) with use(X,E1,P1);

  accepting du;
}
```

Fig. 2. A def-use observer with specifiable variable(s).

2 Input Languages

The modeling language of CO✓ER is based on the model of timed automata [1] or more specifically, the networks of timed automata extended with data variables supported by the UPPAAL tool [6]. For the behavior model and the parsing of the model language CO✓ER benefits from code written for the verifier of the UPPAAL tool. The CO✓ER specific parts are independent of UPPAAL and could be used in similar model-checkers supporting other automata models.

A model of a system often consists of a controller part, specifying the behavior of the system to be tested, and an environment part specifying the components surrounding the controller. We require that the controller part is modeled deterministically so that for a given state and input, a unique response and target state of the controller can be anticipated. This property is called DIEOU-TA (i.e., Deterministic, Input Enabled, and Output Urgent Timed Automata) [3]. The automata in the surrounding environment can be modeled non-deterministically as an ordinary network of timed automata.

The UPPAAL verifier has a query language where a user can specify properties in a subset of Timed CTL, e.g., "exists eventually $P.l_1$" which is true if a state is reachable from the initial state in which automaton P is in location l_1 . The CO✓ER tool extends the language with the prefix `cover` that is used to instruct the tool to generate a test suite which fulfills a coverage criterion specified by an observer.

In CO✓ER, coverage criteria are specified in an observer language [2]. An observer is a monitoring automaton formally describing a coverage criterion. A given observer is refereed to using its name and its parameters, e.g., "cover du({x,y})", where *du* is the name of the observer, and the argument $\{x, y\}$ is a set of variable identifiers in the system model. By convention, CO✓ER in this case assumes that the observer is specified in a file named "du.obs". An example of a coverage criteria described as an observer is shown in Fig. 1. The same observer specified in the input language of CO✓ER is shown in Fig. 2. We refer the reader to [2] for a description of the observer language used in CO✓ER.

A `cover` query can have an optional `restrict` statement which takes a set of automata as argument. When used, the statement instructs the CO✓ER tool to compute new coverage only if at least one of the specified automata are involved in a generated model transition. Our experiments shows that this simple optimization can reduce

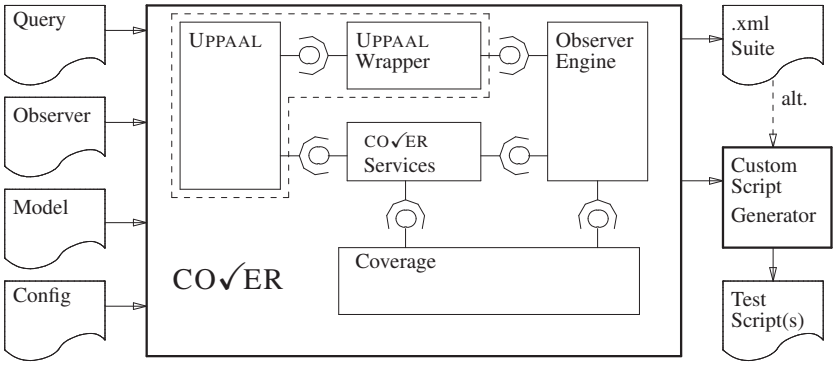

Fig. 3. CO✓ER Architecture and Environment

the time for test suite generation substantially. We have also seen that it can often be applied, since typically only a part of a model specifies the controller to be tested.

3 Tool Overview

In this section, we describe the CO✓ER tool architecture focusing on the main additions to UPPAAL. The work flow of the tool and the internal CO✓ER architecture is shown in Fig. 3. The user specifies a *Model* file and a *Query* file as input. If a query is prefixed by the keyword `cover`, it specifies an *Observer* file to be used. The observer is handled by the *Observer Engine*. The CO✓ER *Services* collects all generated traces that covers part of the coverage criteria, and thus might become part of the test suite. After termination the traces are either passed to an external *Custom Script Generator* or, if a configuration file (*Config*) is specified, they are compiled into a test suite in XML format (*.xml Suite*).

To generate and select test cases, CO✓ER performs state-space exploration by on-the-fly reachability analysis of the timed automata (symbolic) state-space combined with coverage information. During the analysis, a successor state is generated in two steps. In the first step a (symbolic) successor state is generated by UPPAAL. In the second step, the CO✓ER *Services* updates the coverage information attached to the state, with help of the *Observer Engine*. The coverage information is stored and manipulated using bit vector representation managed by *Coverage* in Fig. 3. The interpretation of the bit vectors are known by *Observer Engine* that performs the observer status update.

When the *Observer Engine* calculates observer successors it is dependent on a set of macros used in the observer, to monitor the state and the changes taking place in a a model transition. For example, the observer in Fig. 1 uses the macro *def*(X, E, P), the macro is true with solution $X = x, E = e, P = p$, if variable x is defined in automaton p on edge e and there are no other restrictions on X, E, and P.

In CO✓ER the UPPAAL *Wrapper* translates the UPPAAL model transitions to a generic macro evaluation system used in *Observer Engine*. The interface of *Observer*

Engine can be implemented for other models without modifying *Observer Engine* internally. In this way, CO✓ER can be used to extend other model-checkers or interface with other tools with little effort. In Fig. 3, the parts that are UPPAAL specific are positioned inside a dashed box.

4 Case Study

The CO✓ER tool has been applied in a large case study in cooperation with Ericsson, where a WAP Gateway has been tested [5]. The software of the session layer (WSP) and the transaction layer (WTP) of the WAP stack were modeled in detail. The model also contained automata modeling abstract behavior and assumption imposed on the environment, such as a web sever and terminals using the gateway.

In the case study, we used CO✓ER to produce a test suite with full feasible coverage of edge coverage, switch coverage, and projected state coverage. To perform the actual testing, a complete test bed was built that supports automated generation and execution of tests. It takes as input a network of timed automata, an observer automaton together with the other configuration files, and uses the CO✓ER tool to generate an abstract test suite. The test suite is compiled into a script program that is executed by a test execution environment developed by Ericsson.

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. J. Blom, A. Hessel, B. Jonsson, and P. Pettersson. Specifying and generating test cases using observer automata. In J. Gabowski and B. Nielsen, editors, *Proc. 4^{th} Int. Workshop on Formal Approaches to Testing of Software 2004 (FATES'04)*, volume 3395 of *LNCS*, pages 125–139. Springer–Verlag, 2005.
3. A. Hessel, K. G. Larsen, B. Nielsen, P. Pettersson, and A. Skou. Time-Optimal Real-Time Test Case Generation using UPPAAL. In A. Petrenko and A. Ulrich, editors, *Proc. 3^{rd} Int. Workshop on Formal Approaches to Testing of Software 2003 (FATES'03)*, volume 2931 of *LNCS*, pages 136–151. Springer–Verlag, 2004.
4. A. Hessel and P. Pettersson. A global algorithm for coverage-based test suite generation. In *Accepted for the 3^{rd} Workshop on Model-Based Testing 2007 (MBT07)*.
5. A. Hessel and P. Pettersson. Model-Based Testing of a WAP Gateway: an Industrial Study. In L. Brim and M. Leucker, editors, *Proceedings of 11th International Workshop on Formal Methods for Industrial Critical Systems*, volume 4346 of *LNCS*, pages 116–131. Springer–Verlag, 2007.
6. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.