State Identification and Verification using a Model Checker

Christopher Robinson-Mallett, Peter Liggesmeyer

Fraunhofer IESE
Fraunhofer Platz 1
67663 Kaiserslautern
mallett@iese.fraunhofer.de
peter.liggesmeyer@iese.fraunhofer.de

Abstract: This paper presents a method for the application of model checking, i.e. verifying a finite state system against a given temporal specification, to the problem of generating test inputs. The generated test inputs allow state characterization, i.e. the identification and verification of internal states of the software under test by observation of the input/output behavior only. A test model is derived semi-automatically from a given state based specification and the testing goal is specified in terms of temporal logic. On the basis of these inputs, a model checking tool performs the testing input generation automatically. In consequence, the complexity of our approach is depending on the input model, the testing goal, and the applied model checking algorithm. The presented approach can be adapted with small changes to other model checking tools. It is a capable test generation method, whenever a state based behavioral specification of the software under test exists. Furthermore, it provides a descriptive view on state based testing, which may be beneficial in other contexts, e.g. education and program comprehension.

1 Introduction

Testing is an important, mandatory quality assurance technique in each software development project. The automation of testing potentially reduces human errors as well as time and money efforts. Consequently, it has been an important research topic in the recent years. In this paper, we present a method for the automated generation of test inputs on the basis of state based specifications. Furthermore, we aim at the verification and identification of internal states of the implementation in the case the test is run in black box manner, and therefore observations are restricted to input/output behavior.

Model Checking, i.e. verifying a finite state system against a given temporal specification [CGP2000], has been established in the recent years as a powerful static verification method. The two most prominent approaches to model checking have been introduced independently by Clarke and Emerson [EC1981], based on Computational Tree Logic (CTL), and Quielle and Sifakis [QS81], based on Linear Temporal Logic (LTL). A detailed overview of both approaches can be found in [CGP2000]. In this paper, we will concentrate on the application of the approach by Clarke and Emerson and only a small subset of CTL.

The construction of checking sequences that can be used to check conformance of an implementation against its state based specification has been a major research topic since the earliest days of computing [Mo1956]. The conformance check is based on the derivation of the internal states of the software under test from its input/output behavior. Therefore, a checking sequence is constructed from a *cover set*, e.g. transition cover, and a set of input sequences that are used to characterize each state of the software corresponding to its specification. The construction of cover sets from state based specifications has been under research for many years and seems to be well understood. Recent applications of model checking to the problem of generating test inputs on the basis of coverage criteria bridge the gap between static verification and testing on the basis of state based specifications, e.g. [Hu2004].

In the domain of state characterization problems we briefly discuss the two most prominent: 1) *State Identification*: The problem is to identify the unknown initial state of a state machine under test. 2) *State Verification*: The problem is to verify that a state machine under test is in a particular state at a specific stage of the test. Both problems are addressed with the application of characterization sets. From all the different kinds of characterization sets some efficiently solve the first, e.g. *distinguishing sequences* [Gö70], and others aim at the second, e.g. *Unique Input Output Sequences* [SD1988]. However, any of the presented methods can be applied to both problems, more or less efficiently. In this paper, we follow a more general approach of generating characterization sets without regarding whether state identification or state verification is addressed. In chapter 6 we present an example of a FSM that is used to construct different forms of characterization sets, which can be applied to these problems. In Table 1, Table 2, and Table 3 these resulting characterization sets are presented.

Characterization sets: A characterization set is a set of input sequences, referred to as state characterization sequences, on a minimal finite state machine that produces different output for each different initial state. In this context, the initial state is the state in which the software resides before executing the input sequence. Referring to the example in Section 6 of this paper, a simple form of a characterization set that consists only of the input sequence *baa* decides each state of the finite state machine in Figure 2.

In this paper, we present a method for the application of model checking in order to generate characterization sets that provide full fault coverage [Ch78] on minimal finite state machines. We present a general model and specifications of characterization sets in computational tree logic that can be used in a model checking tool to generate several kinds of characterization sets.

This article is structured into seven sections, including the introduction in Section 1. In Section 2 we discuss related work and position our approach into the research area. In Section 3 the most important basics of state machines and the model checking tool UPPAAL are presented. In Section 4 our approach of generating characterization sets with UPPAAL is presented and its complexity is discussed in Section 5. In Section 6 an example of the application of the presented method to an FSM is given. In Section 7 we conclude this paper and present future research topics.

2 Related Work

The development of automata-theoretic testing methods was originally motivated by checking problems of sequential circuits [Mo1956]. The adoption of these methods to software has been an important research topic over decades. A detailed overview of automata-based testing methods can be found in a number of articles, e.g. [BP94], [SL1989]. Any of these automata-based testing methods are demanding a minimal, complete finite state-machine.

One of the earliest methods of automata-based testing was based on *preset distinguishing sequences* (DS) [Gö70], [He64], [Mo1956]. The major advantage of the DS method is the production of relatively short checking sequences. Sun et. al. presented in [SVJ98] an efficient method for the construction of *unique input output sequences* (UIO) [SD1988], though this method is not generating optimal results. In [Ch78] Chow presented the *W*-method which produces relatively long checking sequences, but which is applicable to each minimal finite state machine.

In [LY1994] Lee and Yannakakis presented a detailed study on the complexity of the construction of DS and UIO, with the negative result that both are PSPACE-complete. Furthermore, not for each minimal FSM a DS or UIO exists. Nevertheless, in our experiments characterization set generation seemed applicable in a large number of cases, mainly depending on the length of the state characterization sequences and the quality of the model checking algorithm.

The generation of DS was already presented in a preceding paper [RMLG2005]. In the same paper we also demonstrated the application of DS generation to finite state machines extended with data and transition guards. Here, we present a general method for the generation of DS, UIOs, *W*-sets from finite state machines.

3 Preliminaries

This section provides a brief introduction to the theory of *finite state machines* (FSM) and to the semantics of the UPPAAL system and requirements description languages.

3.1 Finite State Machine

An FSM is defined by a tuple (S,s_0,X,Y,f_s,f_y), where S is a finite set of *local states* (*locations*), $s_0 \in S$ is the initial state, X is a finite set of inputs, Y is a finite set of outputs, f_s is the state transition function $S \times X \rightarrow S$, f_y is the output function $S \times X \rightarrow Y$. Two states $s_i, s_j \in S$ are called equivalent, if s_i or s_j excited by any input sequence yield identical output sequences. An FSM is minimal, if there is no pair of equivalent states $s_i, s_j \in S$, $i \neq j$. For a detailed introduction to finite state machines and checking problems we recommend the work of Gill [Gi62].

In a system of n parallel FSMs $M_1||M_2||...||M_n$ the *global state* is the combination of each local state of the n state machines M_i.

3.2 UPPAAL Timed Automata

In 1995 the model checker UPPAAL was presented [BLLP95]. It supports an extended version of timed automata [AD94]. Some of the extensions are integer variables and constants, send (*!*) and receive (*?*) synchronization, *urgent* and *broadcast channels*, and *urgent* and *committed locations*. Synchronization over a channel *e* is defined between a sending transition (*e!*) and a receiving transition (*e?*). Synchronization over an urgent channel is preferred to any conflicting synchronization over a not urgent channel. A broadcast channel allows synchronization of multiple automata in one step. A transition, which leaves an urgent state, cannot be delayed and must not possess guard conditions. In a committed location time must not be passing and an outgoing transition must be taken immediately.

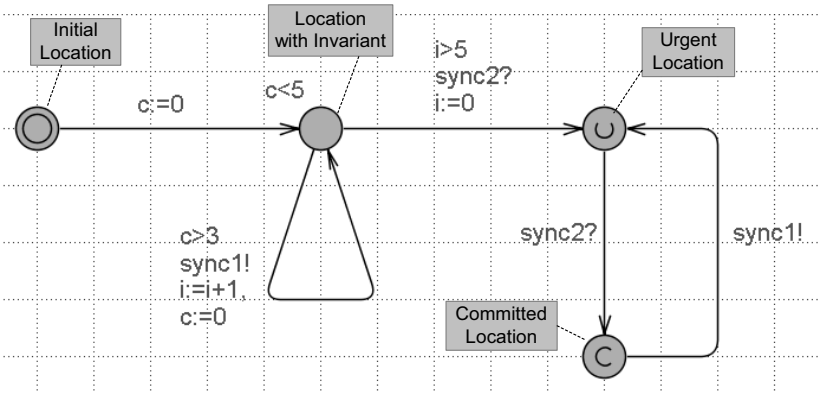

Figure 1: UPPAAL Timed Automaton

The example in Figure 1 presents an UPPAAL timed automaton, which defines locations of the four different types. The transition from the initial state is taken and clock *c* is set to 0. If *c* is greater than 3, the reflexive transition is taken, which synchronizes over channel *sync1*, increments variable *i*, and sets *c* to 0. If *i* is greater than 5 the transition to the urgent location is taken, which synchronizes over channel *sync2* and sets *i* to 0. The transition to the committed location is urgently taken, when it receives a synchronization command over channel *sync2*. The transition that is leaving the committed location is immediately taken, and synchronizes over channel *sync1*.

The model checking tool UPPAAL supports a restricted form of CTL, of which we only need a small subset for the definition of reachability properties. A reachability property describes a state on a trace, in which a certain condition holds. The condition is expressed in terms of propositional logic. The CTL property $E\langle\rangle\ p$ demands the existence of a trace, expressed by E, on which in at least one state, expressed by $\langle\rangle$, the formula p holds.

4 Generating Characterization Sets with UPPAAL

In the following chapter the product and the co-product represent the UPPAAL conjunction operator && and the disjunction operator ||. We prefer this general representation of logical expressions in order to ease the adaptation of our method to other model checking approaches and tools.

4.1 State Characterization Model

In order to generate a characterization set for a FSM M with n local states, a state characterization model $\mathbf{M_s}$ has to be derived from M. In $\mathbf{M_s}=D||M_1||M_2||...||M_n$ for each $s_i \in S$ with $n=|S|$ exists an automaton M_i with s_i as its initial state. Note, the automata M_i are equivalent to M except of the initial states and the data extensions. An automaton D is defined as the representation of a test driver in order to trigger transitions and to evaluate the output products. Each automaton M_i is equipped with an variable out_i, which is initialized with –1 and contains the actual output code on an execution path of M_i. Each transition t_k of M_i is extended with a definition of out_i corresponding to the produced output of t_k. Therefore, each output channel c_i of $\mathbf{M_S}$ is mapped to a unique natural number l_i that is assigned to out_i wherever an output over channel c_i is produced.

In order to evaluate n outputs synchronously on a single input sequence, each automaton M_i must execute synchronously with every M_j in $\mathbf{M_S}$. Therefore, each automaton M_i is synchronously triggered via broadcast communication with D, which also observes and evaluates the output variables out_i. For each pair of output variables $(out_i, out_j),\ i \neq j$ a *Boolean* variable $distinct_{ij}$ is defined, which is initialized with *false* and set *true*, if on an execution path the condition $out_i \neq out_j$ was at least once *true*. After each input and the corresponding outputs, the UPPAAL driver automaton evaluates, whether any of these conditions have become true and updates the corresponding $distinct_{ij}$ variables.

The distinction variable update is expressed in pseudo-code as:

$$\bigcup_{i=0}^{n-2} \bigcup_{j=i+1}^{n-1} if\ out_i \neq out_j\ then\ distinct_{ij} = true$$

It is expressed in the UPPAAL system description language as:

$$\bigcup_{i=0}^{n-2} \bigcup_{j=i+1}^{n-1} (distinct_{ij} := out_i \neq out_j ? true : distinct_{ij})$$

4.2 Distinguishing Sequences

A DS of a minimal FSM *M* consists of an input sequence for the automaton and a unique output sequence for each state of *M*. A state characterization model is used in order to generate a DS, which is complemented with a CTL formula. A DS of *M* exists, if any M_i in $\mathbf{M_s}$ produces a different output on the same input. Therefore, each variable $distinct_{ij}$ must become *true* on an execution path, which is a DS. A single CTL formula is used to generate a DS as follows:

$$\text{F1: } E <> \prod_{i=0}^{n-2} \prod_{j=i+1}^{n-1} distinct_{ij}$$

The UPPAAL model checker produces a (shortest) trace *t* once F1 holds on a search path. An input/output sequence can be constructed easily from *t* and *M*. A shortest trace, which is produced by UPPAAL in response to F1 on *M*, can be transformed into an optimal DS of *M*.

4.3 Unique Input Output Sequences

A UIO verifies a specific state *s* of an FSM *M*, unless an equivalent state to *s* exists in *M*. Therefore, a minimal FSM with *n* states demands *n* UIOs for the identification of each state. In order to generate these UIOs, a state characterization model is used, which is complemented with a specification in terms of CTL. For each state s_i, a specific CTL formula demands that M_i produces distinguishable output to any other M_j, where i≠j, and therefore the variables $distinct_{ij}$, note that $distinct_{ij}=distinct_{ji}$, become *true*. For an FSM with *n* states the UIOs are generated with the following set of formulae:

$$\text{F2: } UIO_M := \bigcup_{i=0}^{n-1} E <> \prod_{j=0}^{n-1} (i = j) \vee distinct_{ij}$$

4.4 W-Sets

A *W*-set contains a number of input sequences, which as a whole identify each state of a minimal FSM *M* when executed. A state characterization model is used in order to generate a *W*-set. The generation of a *W*-set demands the definition of a CTL formula for each pair of states (s_i,s_j), which demands that M_i produces distinguishable output to M_j, where $i \neq j$, and therefore the variable $distinct_{ij}$ becomes *true*. In a first step the *W*-set is generated with the following set of formulae:

$$\text{F3: } W := \bigcup_{i=0}^{n-2} \bigcup_{j=i+1}^{n-1} E <> distinct_{ij}$$

In a second step all input sequences are eliminated from the set, which are contained as a starting sequence of another element in the same set.

F4: $W := \{s \in W \mid \neg\exists_t st \in W\}$

5 Complexity

The complexity of CTL model checking has been discussed in a number of publications, e.g. [CGP2000]. Here, we will briefly discuss the complexity of generating DS, UIOs, and *W*-sets with UPPAAL under the assumption that *breadth first search* is used and no complexity reduction methods are applied.

The complexity of the presented approach of generating *characterization sets* with the UPPAAL model checker, is depending on the input model, the characterization problem, and the model checking algorithm. A *state characterization model* does not imply a higher time complexity than the FSM, from which it is derived. Although, an FSM *M* is multiplied *n* times, where *n* is the number of states in *M*, and additional variables are needed, none of these modifications increase the time consumption of the model checking problem. Only the space consumption is raised, due to the enlarged state representation, which is linear to the number of states *n* of *M*. It is well known that time complexity of a reachability problem under *breadth first search* is exponential to the length of the required path, which is in this case a shortest path. The length of a shortest *state characterization sequence* is depending only on the automata structure. Also, time consumption is polynomial to the size of the input alphabet, which is only depending on *M*.

The time complexity of our approach of generating a state characterization sequence is independent of the number of states, polynomial to the size of the input alphabet, and exponential to the length of the shortest state characterization sequence.

The generation of a single UIO is of the same complexity as the generation of a DS. Taking into account, that UIOs are generated for each state of a machine, time consumption is linear to the number of states. The time consumption of the generation of a *W-set* is polynomial ($O(n^2)$) to the number of states *n*.

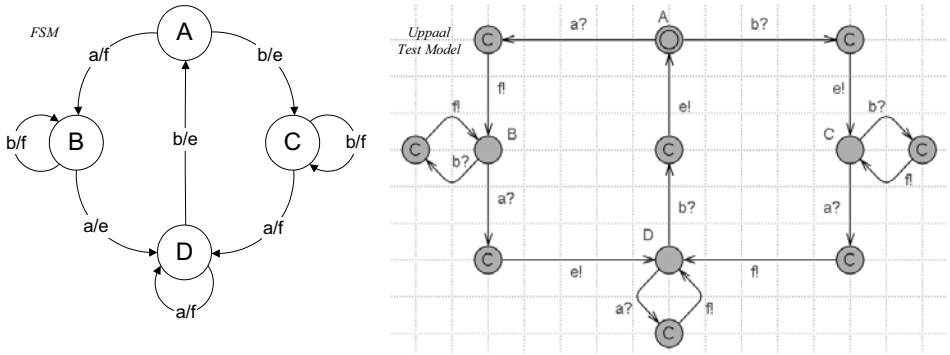

Figure 2: FSM Test Model

The generation of UIOs is based on weaker conditions than DS generation. Also, the criteria used for W-set generation are weaker than those for UIO and DS. This indicates that there might be a less time consuming solution for a W-set compared to UIO and for UIO compared to DS. Obviously, these indications do not decide, which method is generally the most efficient.

6 Example

In Figure 2, Figure 3, and Figure 4 the example of a *state characterization model* of an FSM is presented. The following sections describe the generation of DS, UIOs, and W-sets on the basis of the same characterization model, derived from the FSM in Figure 2.

The FSM in Figure 2 contains the four states *A*, *B*, C, and *D*. The characterization model is constructed by multiplication of the FSM corresponding to the number of states and mapping of the outputs *e* and *f* to data values. Corresponding to the states *of* the FSM, in the characterization model exist the four automata M_A, M_B, M_C, and M_D with the initial states *SA*, *SB*, *SC*, and *SD*. The characterization model contains the output variables *A*, *B*, *C*, and *D* that hold the actual output values of the corresponding automata. The outputs *e* and *f* of the FSM are mapped to the data values 0 and 1. For the inequations $A \neq B$, $A \neq C$, $A \neq D$, $B \neq C$, $B \neq D$, $C \neq D$ corresponding boolean distinction variables *AB, AC, AD, BC, BD, CD* are defined. A distinction variable *XY* is initially defined *false* and will permanently become *true* once the inequation $X \neq Y$ becomes *true* on a trace. The evaluation of the output variables and the definition of the distinction variables are performed in the driver automaton.

The *state characterization model driver* in Figure 3 contains an initial committed location and an urgent location. In the initial, committed location the transitions to the urgent location with the sending synchronizations *a*! and *b*! are enabled, and non-deterministically triggered. The transitions that are leaving the initial, committed location are synchronously triggered with at most a single transition in each automaton of the state characterization model. The driver stays in the urgent location until each transition starting in a committed location in the *state characterization model* has been executed. Afterwards, the output evaluation and distinction variable definition is performed.

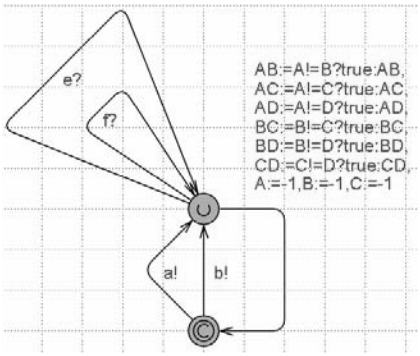

Figure 3: State Characterization Model Driver

6.1 DS Generation

A single characterization property is constructed by application of formula F1 in order to generate a distinguishing sequence. Therefore, a distinguishing sequence exists, if the following property becomes true on the state characterization model:

E<> AB&&AC&&AD&&BC&&BD&&CD

The UPPAAL model checking tool delivers a shortest trace on that the property becomes true. From this trace the input sequence *baa* is extracted, which is a shortest DS for the presented example. The characterization of states by application of the generated DS to the FSM is demonstrated in table 1.

Table 1: DS of Figure 2

input / initial state	***baa***
A	*eff*
B	*fef*
C	*fff*
D	*efe*

Table 2: UIO of Figure 2

initial state	***i/o***
A	*aa/fe*
B	*a/e*
C	*ba/ff*
D	*bb/ee*

Table 3: W of Figure 2

initial state	***a***	***b***	***aa***
A	*f*	*e*	*fe*
B	*e*	*f*	*ef*
C	*f*	*f*	*ff*
D	*f*	*e*	*ff*

6.2 UIO Generation

The generation of a UIO sequence for each state of the given example follows formula F2. The complete set of UIO sequences exist, if the following properties become true:

E<> AB&&AC&&AD; E<> AB&&BC&&BD;
E<> AC&&BC&&CD; E<> AD&&BD&&CD

The UPPAAL model checker delivers a shortest trace for each query. Table 2 demonstrates the application of the generated UIOs on the example in Figure 2.

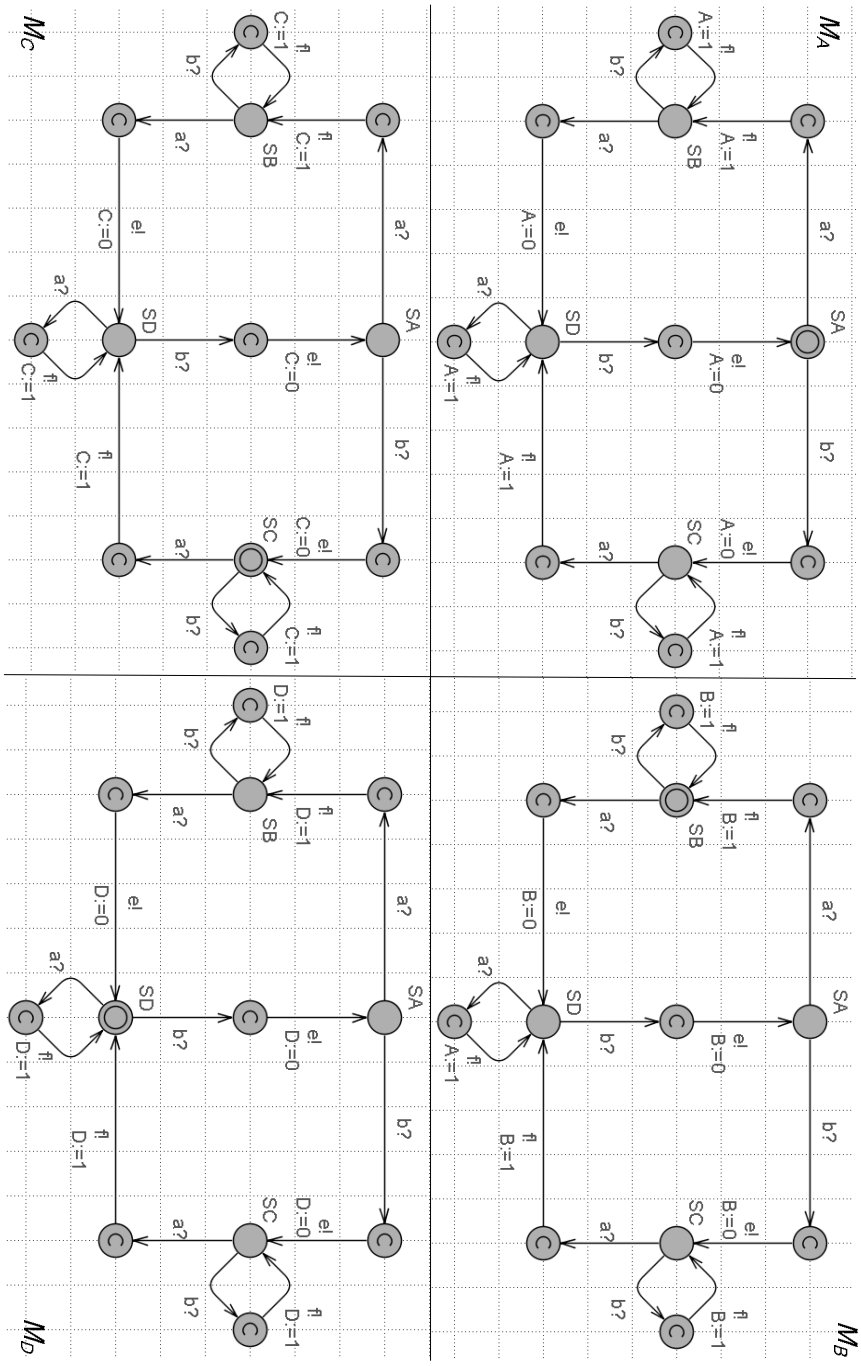

Figure 4: State Characterization Model

6.3 W-set generation

The construction of a *W*-set follows formulae F3 and F4. In a first step the UPPAAL model checker produces a set of input sequences, which characterize each state as a whole.

E<> AB; E<> AC; E<> AD; E<> BC; E<> BD; E<> CD

The UPPAAL model checking tool produces a shortest trace for each formula. These traces already form a valid *W*-set, which possibly contains redundant elements corresponding to formula F4. Therefore, in a second step the redundant traces are eliminated. The *W*-set is defined as follows:

W={b,ab}

The results of W-set generation are presented in Table 3, where the column with redundant input *a* is filled grey.

7 Conclusion

Model checking and automatic test-case generation have proven useful in many application areas. In this paper we presented an approach for the automated generation of test inputs using model checking. The generated test inputs allow the identification and verification of internal states of the software under test corresponding to the state based specification that is used for test generation. The ability to derive the internal state of the software under test by input/output observation only is especially in those cases beneficial, when black box testing is performed.

We presented an approach for automated generation of *preset distinguishing sequences*, *unique input output sequences*, and *W*-sets with the model checker UPPAAL. We have shown that model checking is an appropriate method for generating checking sequences for finite state machines. Furthermore, it can be applied to finite state machines extended with bounded integers and data, as presented for the generation of DS in [RMLG2005].

The presented approach can be adapted to other model checking tools in order to generate test inputs. The problem of generating a characterization set is reduced to the definition of a state characterization model and a specification in terms of temporal logic that describes the specific characterization set. The generation of the characterization set is automatically achieved by the model checking tool. For these reasons the complexity and the capability of our approach are mainly depending on the used model checking tool. This includes the generation of optimal results depending on the ability of the model checker to produce shortest traces. The complexity of our approach corresponds to the results, which were presented by Lee and Yannakakis [LY1994].

For future work we are planning the application of this general approach to finite state machines extended with data and transition guards. Furthermore, the ability of UPPAAL to handle timed specifications we will apply our approach on timed state characterization problems [ENN02]. In order to achieve better performance of our approach, we plan to adapt this method to heuristic and symbolic model checking approaches. The application of complexity reduction methods, e.g. data abstraction [CGP2000], is not extensively explored but will be part of our future work. Furthermore, we know only little about the practical applicability. For the near future we are planning to perform a number of industrial and experimental case studies in order to validate the presented approach.

References

[AD94] Alur R.; Dill L.: A Theory of Timed Automata. Theoretical Computer Science, 126(2):183–235. 1994.

[BLLP95] Bengtsson J.; Larsen K. G.; Larsson F.; Pettersson P.; Yi W.: UPPAAL - A Tool Suite for Automatic Verification of Real-Time Systems, Workshop on Verification and Control of Hybrid Systems, DIMACS, 1995

[BP94] Bochmann G. v.; Petrenko A.: Protocol Testing: Review of Methods and Relevance for Software Testing. Proc. ISSTA'94, pages 109–124. ACM Press. 1994.

[Ch78] Chow T.S.: Testing Software Design Modeled by Finite-State Machines. IEEE Transactions On Software Engineering, 4, Nr. 3. 1978.

[CGP2000] Clarke M.; Grumberg O.; Peled D. A.: Model Checking. MIT Press. Boston. 2000.

[EC1981] Clarke E.M.; Emerson E.A.: Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. Workshop on Logic of Programs. Yorktown Heights, NY. LNCS 131, Springer Press. 1981.

[ENN02] En-Nouaary A., Dsoulli R., Khendek F.: Timed Wp-Method: Testing Real-Time Systems. IEEE Transactions on Software Engineering. Vol. 28 (11). November 2002

[Gi62] Gill A.: Introduction to the Theory of Finite-State Machines. McGraw Hill. Berkeley. 1962

[Gö70] Gönenc G.: A Method for the Design of Fault Detection Experiments. IEEE Transactions on Computers, C-19, June, Nr. 6. 1970.

[He64] Hennie F.C.: Fault-Detecting Experiments for Sequential Circuits. Symposium on Switching Circuit Theory and Logical Design NJ. 1964.

[Hu2004] Huhn M.; Mücke T.: Generation of Optimized Testsuites for UML Statecharts with Time. Testing of Communicating Systems (TestCom'04). Oxford. Springer. 2004.

[LY1994] Lee D.; Yannakakis M.: Testing Finite-state Machines: State identification and verification. IEEE Transactions on Computers, 43(3). 1994.

[Mo1956] Moore E. F.: Gedanken-Experiments on Sequential Machines. Automata Studies (Annals of Mathematics Studies), 34. 1956.

[QS81] Quielle J.P.; Sifakis J.: Specification and Verification of Concurrent Systems in CESAR, Proc. 5th Intern. Symp. on Programming. 1981

[RMLG2005] Robinson-Mallett C.; Mücke T.; Liggesmeyer P.; Goltz U.: Generating Optimal Distinguishing Sequences with a Model Checker. Advances in Model-Oriented Software Testing (A-MOST'05). St. Louis. 2005.

[SD1988] Sabnani K.; Dahbura A.: A Protocol Test Generation Procedure. Computer Networks and ISDN Systems, 15. North-Holland. 1988.

[SL1989] Sidhu D.; Leung T.-K.: Formal Methods for Protocol Testing: A Detailed Study. IEEE Transactions on Software Engineering, 15(4). 1989.

[SVJ98] Sun D.; Vinnakota B.; Jiang W.: Fast State Verification. Conference on Design Automation. ACM Press. 1998.