Achieving Communication Coverage in Testing

Christopher Robinson-Mallett
Fraunhofer IESE
Kaiserslautern
Germany
mallett@iese.fraunhofer.de

Robert M. Hierons
Brunel University
Uxbridge
United Kingdom
rob.hierons@brunel.ac.uk

Peter Liggesmeyer
University of Kaiserslautern
Germany
liggesmeyer@informatik.uni-kl.de

ABSTRACT

This paper considers the problem of testing the communication between components of a timed distributed software system. We assume that communication is specified using timed interface automata and use computational tree logic (CTL) to define coverage criteria that refer to send- and receive-statements and communication paths. Given such a state-based specification of a distributed system and a concrete coverage goal, a model checker is used in order to determine the coverage provided by a finite set of test-cases, expressed using sequence diagrams. If parts of the specification remain uncovered then a goal is derived so that the model checker can be used to generate test cases that increase the coverage provided by the test suite. A major benefit of the presented approach is the generation of a potentially minimal set of test cases with the confidence that every interaction between components is executed during testing. A potential additional benefit of this approach is that it provides a visual description of the state based testing of distributed systems, which may be beneficial in other contexts such as education and program comprehension. The complexity of our approach strongly depends on the input model, the testing goal, and the model checking algorithm, which is implemented in the used tool. While a particular model checker, UPPAAL, was used, it should be relatively straightforward to adapt the approach for use with other CTL based model checkers.

Categories and Subject Descriptors

F.1.1 automata, F.2.2 computations on discrete structures, D.2.5 testing tools D.2.4 model checking

General Terms

Measurement, Design, Reliability, Languages, Theory, Verification.

Keywords

Conformance Testing, Distributed Systems, Coverage Criteria, Timed State-Based Specifications

1. INTRODUCTION

Testing is one of the most frequently used quality assurance techniques in practical software development. Motivation for the practical deployment of software testing techniques ranges from the location of as many defects as possible to reliability analysis. Test automation limits the scope for human errors and can significantly reduce the cost and time involved in software development. Consequently, it has been an important research topic in recent years. In this paper, we present a method for the automated generation of test inputs for distributed systems on the basis of state based specifications whose semantics can be described in terms of interface automata [3].

The construction of test 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 [22]. The problem of constructing a test suite that achieves a given level of coverage of a state-based specification is relatively well understood (see, for example, [23]). Recently model checkers have been used to automate test generation on the basis of a coverage criterion, bridging the gap between static verification and testing (see, for example, [17] or [26]).

Here, we address the problem of generating test-cases for real-time distributed systems, where the tests provide a certain form of coverage. Our approach resolves timing and feasibility problems both on the component and on the inter-component level, which frequently occur when timed state-based specifications are used for system behavior specification. The software under test is assumed to consist of several black-box components. The complexity of the overall system grows exponentially as the number of components increases and thus it is usually infeasible to generate a single model for the system. Since we use a centralized test architecture we avoid the controllability and observability problems described by Cardell-Oliver [7] and Khoumsi [19]. With the proposed communication coverage criteria we perform coverage analysis and test-case generation using the test-models as an input to a model checker. The focus is on covering the possible communications between machines, as represented by the sending and receiving of messages. We define two coverage criteria, the weaker insisting that all possible send/receive pairs are covered. The stronger criterion recognizes that the messages will be sent through an underlying network and there may be different paths through this network: it insists that for each send/receive pair we use every possible path through the network.

Definition

sr-pairs coverage: A combination of send (s) and receive (r) statements for a message m in different machines is called an sr-pair. In order to achieve full sr-pairs coverage every sr-pair must be executed at least once in testing.

Definition

sr-paths coverage: When a message m is exchanged it passes through a communication path p in a network that starts with a send statement s and ends with a receive statement r. Coverage is complete if every communication path is executed at least once in testing.

Note that since a communication path starts with a send statement s and ends with a receive statement r, full sr-paths coverage subsumes full sr-pairs coverage. Coverage analysis and coverage-based test-case generation are performed using a model checker. Model Checking, i.e. verifying a finite state system against a given temporal specification [9], has been established in recent years as a powerful static verification method. The two most prominent approaches to model checking were introduced independently by Clarke and Emerson [10], based on Computational Tree Logic (CTL), and Queille and Sifakis [25], based on Linear Temporal Logic (LTL). A detailed overview of both approaches can be found in [9]. In this paper, we will concentrate on the application of the approach by Clarke and Emerson and use only a small subset of CTL.

In [3], Alfaro and Henzinger propose the use of parallel finite automata in order to describe communication between components in a distributed system. In [4] they expand their approach to timed specifications. Alfaro and Henzinger consider two components to be compatible if there is some environment in which they work correctly and they locate the test-generation and coverage analysis problems into the domain of gaming problems. We are interested in the problem of testing from such interface automata in order to test the communication between components.

We assume that a system description is given in the semantics of open interface automata. During coverage analysis we transform the open interface automata into a closed system by adding a (non-deterministic) tester component that interacts with the system components. We show how a testing goal can be defined in CTL and how interface automata can be transformed into the input language of the model checker. The model checker searches the state space for a sequence that fulfils the (partial) testing goal.

A major benefit of the presented approach is the generation of a potentially minimal set of test cases with the confidence that every interaction between components is executed during testing. The presented approach is applicable to a variety of practical testing problems, where state-based specifications with data extensions and timing constraints are used to describe the behavior of distributed real-time systems. Additional value can be found in the visual description of the state based testing of distributed systems, which may be beneficial in other contexts such as education and program comprehension.

This article is divided into seven sections, including the introduction in Section 1. In Section 2 an example of a distributed system is presented. In Section 3 the most important basics of state machines, interface automata, and timed automata as used in UPPAAL are presented. In Section 4 we reason about test coverage on the inter-component level and propose a new coverage criterion. In Sections 5 and 6 we present methods for test coverage analysis and test generation on the basis of the presented coverage measure. In Section 7 we discuss complexity issues of our approach. In Section 8 we discuss related work and position our approach into the research area and in Section 9 we conclude this paper and present future research topics.

2. Example

An example of a sorting line is presented in Figure 1. The distributed system consists of three software controlled components. On a production line objects, here triangles and squares, pass a sensor. The sensor signals that an object has passed, so that this object will pass the sorter in 3 seconds. The sorter checks the object. If it passes the test, it is pushed into the basket of good objects. If it fails, it drops at the end of the line into the basket of bad objects. If an object enters the sensor while another object is still being processed, the whole system stops.

An abstract description of Sensor, Sorter, and Counter is presented in Figure 2. The semantics of the behavior specification are those of a state-machine annotated with textual descriptions of the timing constraints and the triggering conditions. In this model the variable res represents the result of the sorter's check on the object: it is true if the object passes the test.

Furthermore, an abstract specification of the required communication between Sensor, Sorter, and Counter is presented in Figure 3 for the case that a "good" object is processed.

We assume that the underlying reliable communication system provides a worst case delivery time of 1s.

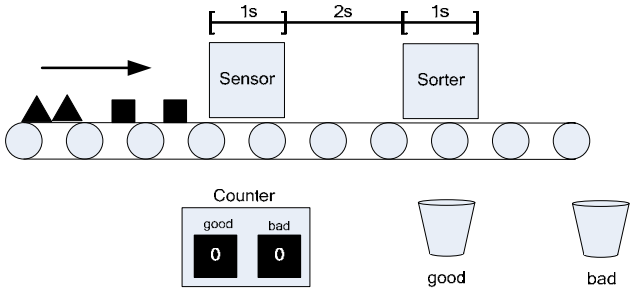

Figure 1. Example of an Distributed System

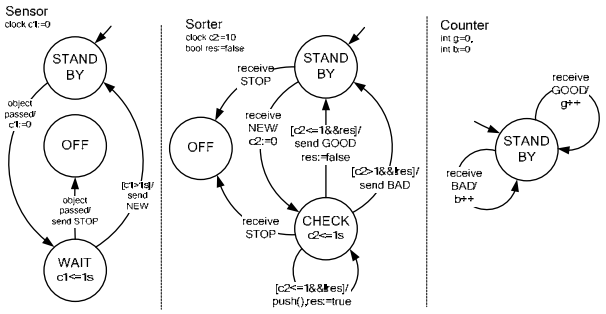

Figure 2. Behavior of Example in Figure 1

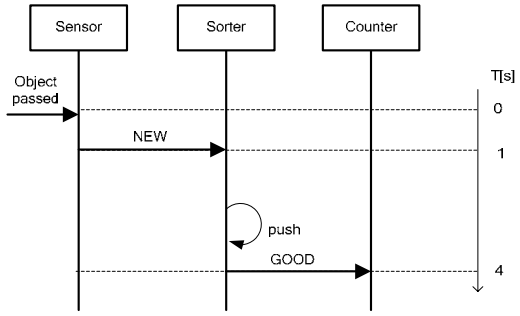

Figure 3. Communication for object classified "GOOD"

The example will be used throughout the paper in order to demonstrate communication coverage analysis of a given test set and the generation of test cases for certain criteria. Briefly, the testing problems can be described as follows:

A. Definition of a complete set of communication paths between components. The example in Figure 2 contains several communication paths, such as that defined by Sensor sending the message NEW through a transition from state WAIT and Sorter receiving this message through a transition from state STAND BY.

B. Applying the test inputs (Figure 3) to the model of parallel, communicating state-machines (Figure 2). This step can be performed manually or automatically, automation having the advantages that it reduces the time involved and the scope for errors. In this paper we describe the use of a model checker.

C. Checking, which communication paths have not been tested. In the example we can immediately determine, for example, that the STOP signal between Sensor and Sorter has not been used. However, this step is hard to perform manually, except for the most trivial cases. For a complete analysis of missing executions we refer to Section 5.

D. Generation of test-cases that execute the missing communication paths, which were identified in step C, or determining that these are infeasible. Similar to the preceding step test case generation is hard to perform manually once models exceed a certain complexity. In Section 6 we discuss how a complete tests suite can be produced for the example.

3. PRELIMINARIES

This section provides a brief description of *finite state machines* (FSMs) and extended finite state machines (EFSMs) in order to introduce the notions of a state and an extended state. Interface Automata are introduced for their ability to describe the visible behavior of a distributed system in a straightforward and generic fashion. UPPAAL Timed Automata (UTA) and CTL serve as input languages to the model checker. Therefore we use the syntax of UTA in order to describe timed interfaces and express coverage criteria in terms of CTL.

3.1 Finite State Machine

An FSM is defined by a tuple (S,s_1,X,Y,h), where S is a finite set of local states (locations), $s_1 \in S$ is the initial state, X is a finite set of inputs, Y is a finite set of outputs, and h is the transition function $S \times X \rightarrow P(S \times Y)$, where $P(S \times Y)$ denotes the powerset of $S \times Y$. If $(s',y) \in h(s,x)$ and we input x when the FSM is in state s then the FSM can move to state s' and output y. Two states $s_i, s_j \in S$ are said to be equivalent, if for every input sequence x the set of output sequences that can be produced by applying x when in state s_i is the same as the set of output sequences that can be produced by applying x when in state s_j. 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 FSMs and testing from FSMs we recommend the work of Gill [11].

In this context, a parallel composition $M=M_1 \| M_2$ describes independently executing FSMs $M_1=(S_1,s_1,X,Y,h_1)$ and $M_1=(S_2,s_2,X,Y,h_2)$. The semantics of M are represented by the product of M_1 and M_2, i.e. $M=(S_1 \times S_2,(s_1,s_2),X,Y,h)$, where $((s_m,s_n),y) \in h((s_i,s_j),x)$ if and only if $(s_m,y) \in h_1(s_i,x)$ and $(s_n,y) \in h_2(s_j,x)$.

In a system of n parallel FSMs $M=M_1 \| M_2 \| \ldots \| M_n$ a global state, also referred to as a configuration of M, represents the combination of each local state of the n individual state machines.

FSMs may be deterministic or non-deterministic. An FSM is deterministic if for every state s and input x we have that h(s,x) contains at most one element: there is no state s such that two or more transitions leaving s have the same input symbol.

3.2 Extended Finite State Machine

An EFSM is defined by a tuple $(S, s_1, v, v_1, X, Y, T)$, where S is a finite set of logical states, $s_1 \in S$ is the initial state, v is a tuple of bounded integer values, v_1 is a tuple of initial values for v, X is a finite set of input symbols, Y is a finite set of output symbols, and T is a finite set of transitions. In the following we let V denote the set of possible values for the tuple v and so $v_1 \in V$. A transition is defined by a tuple (s_s, x, g, u, y, s_f), where $s_s \in S$ is the local start state, x is an input in X, g is a boolean guard function of type $V \rightarrow \{true,false\}$, u is a variable update function of type $V \rightarrow V$, y is an output from Y, and $s_f \in S$ is the local target state. A transition can be triggered, if and only if its guard is true.

An extended state of an EFSM M, also referred as the configuration of M, represents a tuple of a local state and a value of v drawn from V. Each EFSM can be transformed into an FSM, where a state is represented as a tuple of its local state and a vector of data values.

3.3 Interface Automata

The definition of interface automata is taken from the work of Alfaro and Henzinger [3], which we recommend for a detailed study.

An interface automaton is defined by a tuple $P=(V_P,V_P^{init},A_P^I,A_P^O,A_P^H,T_P)$, where V_p is a finite set of states, $V_P^{init}\subseteq V_P$ is a set of initial states. We require that V_P^{init} contains at most one state. If $V_P^{init}=\emptyset$, then P is said to be empty. A_P^I, A_P^O, and A_P^H are mutually disjoint sets of input, output, and internal actions. We denote by $A_P=A_P^I\cup A_P^O\cup A_P^H$ the set of all actions, $T_P\subseteq V_P\times A_P\times V_P$ is a set of steps.

An execution fragment of an interface automaton P is a finite alternating sequence of states and actions $v_0,a_0,v_1,a_1,...,v_n$ such that $(v_i,a_i,v_{i+1})\in T_P$ for all $0\le i<n$. Interface automata are not required to be input-enabled [3]; in a state there might be input in A_p^I that are not accepted. If an interface automaton only contains internal actions we say it is closed; otherwise it is open.

A composition of interface automata is only defined if their actions are disjoint, except that an input action of one may coincide with an output action of another. Two interface automata will synchronize on such shared actions, and asynchronously interleave all other actions. The composition of interface automata P and Q is the product automaton P⊗Q, see [3] for a detailed definition. Since P and Q are not necessarily input enabled [3], some steps present in P and Q may not be present in P⊗Q and there may exist illegal states in P⊗Q in which P can produce an output that is not accepted by Q.

A nonempty interface automaton E is a *legal environment* for P and Q, if P⊗Q is open and P⊗Q⊗E is free of illegal states, and it holds that $A_E^I=A_{P\otimes Q}^O$. Two nonempty interface automata P and Q are compatible if there exists a legal environment for P⊗Q. Two states in different interface automata P and Q are said to be compatible if an environment can prevent them from entering an illegal state.

3.3.1 Timed Interface Automata

The following definitions are taken from [4] where Alfaro et al. extend their interface theory to timed models.

An interface automaton is defined by a tuple $P=(V_P,V_P^{init},A_P^I,A_P^O,R_P^I,R_P^O)$, where V_p is a finite set of states, $V_P^{init}\subseteq V_P$ is a set of initial states. A_P^I and A_P^O are mutually disjoint sets of immediate input and output actions. Furthermore, A_P^I and A_P^O are disjoint from the set of timed actions T. We denote by $T_P^I=A_P^I\cup T$ the set of all input actions, and by $T_P^O=A_P^O\cup T$ the set of all output actions. $R_P^I\subseteq V_P\times T_P^I\times V_P$ is the input transition relation and $R_P^O\subseteq V_P\times T_P^O\times V_P$ is the output transition relation.

In contrast to the requirements of Alfaro et al. in [4], we do not require the transition relations to be functions and so we allow non-determinism.

3.3.2 UPPAAL Timed Automata

In 1995 the model checker UPPAAL was presented [5]. It supports an extended version of timed automata 0. 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 channel that is not urgent. 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 pass and an outgoing transition must be taken immediately.

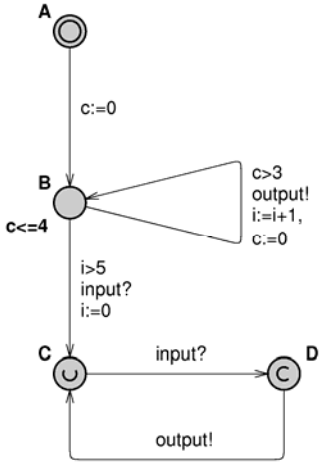

Figure 4. UTA Example

The example in Figure 4 presents an UPPAAL timed automaton (UTA), which defines locations of the four different types. The sending and receiving of messages are mapped to the synchronous operators (!) and (?). The transition from the initial location A to location B is taken and clock c is set to 0. Execution remains in B unless c exceeds 4. If c is greater than 3, the reflexive transition (B→B) is triggered, which sends over channel "output", increments variable i, and resets c to 0. If i is greater than 5 and "input" is received the transition to the urgent location C is taken and i is reset to 0. The transition to committed location D is urgently triggered, when "input" is received. The transition from D to C is immediately triggered and sends "output".

The model checker 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<> p demands the existence of a trace, expressed by E, on which in at least one state, expressed by <>, the formula p holds.

3.4 Test Model

In order to analyse test coverage and generate test cases with the model checker a test model in the form of UTA is constructed from the abstract behavior description. For a detailed description of test coverage modeling using UTA we recommend [17] and [26].

The UTA behavior descriptions of Counter, Sensor, and Sorter in Figures 5-7 are almost identical to the models given in Figure 4. In contrast to the abstract behavior, in the test model we need a model of the component "BUS" (Figure 8) that represents the message transportation delay through variable dla. For communication between Counter, Sensor, and Sorter the delay is at most 1s. The signals NEW, GOOD, BAD, and STOP are split into input (i) and output (o) sections that are relayed through

BUS. In order to represent upper and lower bounds of message transportation delay, for each output BUS contains an unconstrained sending transition and one with a 1s delay guard.

Each component possesses its own clock, which is used in state invariants and transition guards in order to formalize the timing properties.

Recall that the Boolean variable res is set to true when a "good" object is checked. The value of this variable determines whether the transition sending the "GOOD"-message or the transition sending the "BAD" message is triggered.

4. COVERAGE CRITERIA FOR PARALLEL TIMED AUTOMATA

Typical coverage criteria, e.g. state coverage or transition coverage, work well if we only deal with single automata. When we have many parallel machines these criteria might not be sufficient. This is both because they can lead to a combinatorial explosion and because they do not explicitly consider the possible interactions between the machines.

The principles for the specification of communication in a timed distributed system have been described in [3] and [4]. This work has also shown that the problem of checking consistency between a specified communication, e.g. sequence diagrams, and the behavior of a distributed system, e.g. parallel timed automata, can be described as a gaming problem. Briefly, a message sequence σ, which is a word, is feasible for a system behavior when the product automaton of all automata of the system accepts σ. The question as to whether or to what degree a set of message sequences covers the product automaton, and therefore the possible communication, is also of our concern. It can easily be answered using static analysis, for instance by using model checking.

Here, we don't have the product automaton at hand, but the model checker will produce parts of this during the search. In order to measure the coverage provided by a set of message sequences we have to relate the structure of the (unknown) product automaton to the automata of the system. We know that each send and receive pair in parallel automata may cause a transition in the product automaton. Therefore we can define a weak coverage criterion for send and receive pairs, referred to as *sr-pairs-coverage*. It is weak in the sense that due to transition guards and automata structures we cannot guarantee that each sr-pair is feasible, and thus we may never obtain complete sr-pairs coverage.

In addition to sr-pairs coverage we might also demand transition coverage. This is feasible when all transitions are executable but potentially leads to unnecessarily many test-cases as a result of the combinatorial explosion that occurs when the product automaton is produced. Transition coverage applies to parallel automata as described in [17], therefore we recommend this work for a detailed description.

Here we will briefly describe the application of sr-pairs coverage to the given example.

The coverage of sr-pairs is measured through adding variables to the test model. For each send and receive statement one Boolean variable is needed that is set to true directly after the execution of the corresponding statement. Furthermore, for each sr-pair one Boolean variable is needed, which is set true once both the corresponding send and receive variables are true. After updating the pair of variables all send and receive variables are reset to false.

Figures 5-8 show the UPPAAL models corresponding to those in Figure 2 with assignments added in order to record the sr-pairs executed. In this example, for each message we have only one send statement and only one receive statement. It is thus sufficient to add only one variable for each message but in general we need one variable for each send statement and one variable for each receive statement. For example, in Figure 5 we see that if NEWi? is received then sr[0] is set to true but there is no need to record the execution of the sending transition, NEWo!, in Figure 6 since this is the only transition that can send this message.

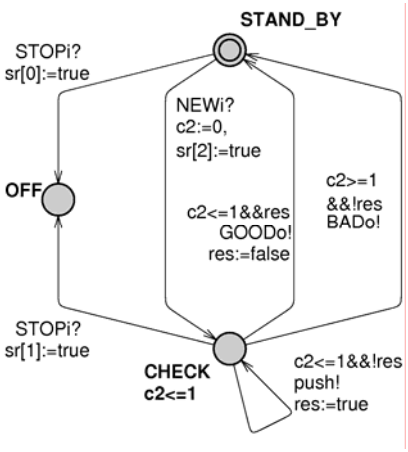

Figure 5. Test model component "Sorter"

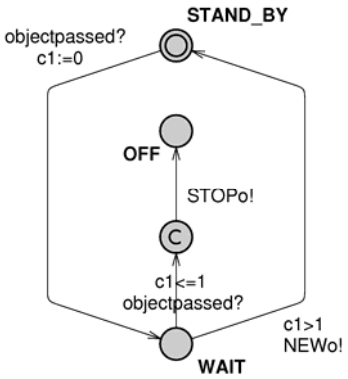

Figure 6. Test model component "Sensor"

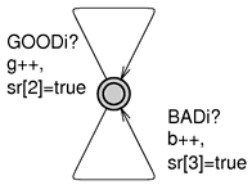

Figure 7. Test model component "Counter"

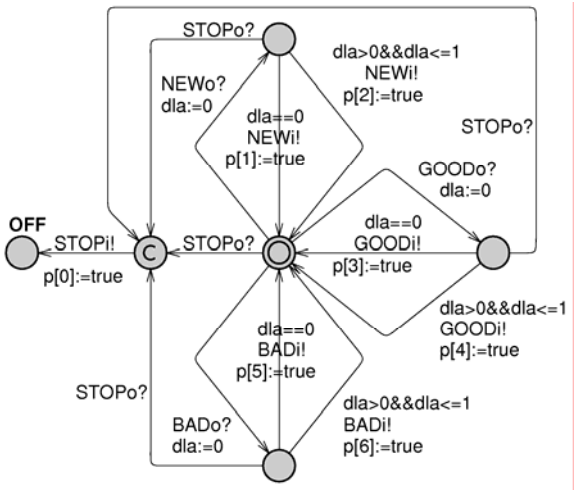

Figure 8. Test model component "BUS" of communication system

5. TEST COVERAGE ANALYSIS

Earlier in this paper we introduced the communication path coverage (CPC) and sr-pairs coverage (SRPC). The relation between these criteria can be described as follows: If communication is realized as a peer-to-peer (2-machines) or as a centralized communication network, and thus each communication is realized through a single path only, CPC equals SRPC. If communication is organized in a decentralized manner, e.g. peer-to-peer (>2-machines), and thus a message may take several routes through the network, CPC subsumes SRPC. The characteristics of the underlying communication network can be introduced into the test model through an additional relay component. In the example in Figure 8 the delay of 1s of the communication system is introduced into the test model through such a relay component. For testing we might consider worst case delivery time, where each message is delayed by 1s, and best case delivery time, where messages are not delayed. These cases are represented in the BUS component through corresponding transitions. In the example communication is centralized and so we concentrate on SRPC in the remainder of this paper.

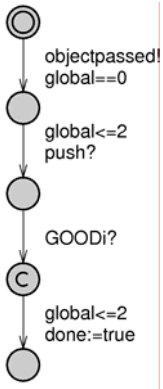

Figure 9. Test case component for object classified "GOOD"

The test coverage analysis is performed on a test model, which is extended with coverage variables. An array of the size of the number of sr-pairs and an array of the size of the number of communication paths are defined. An element of the coverage arrays is defined true, once the corresponding sr-pair or path is executed by a test-case. The test case component "GOOD CASE" in Figure 9 is a UTA that corresponds to the test-case in Figure 3 and this can be executed on the test model. The Boolean variable done indicates that the test-case has been executed completely.

Given the example of a test model in Figure 5, Figure 6, Figure 7, Figure 8, and Figure 9 we briefly demonstrate coverage analysis. According to the test model and its instrumentation variables the execution of the test cases, using the CTL property E<> done, causes the following pair variable configurations for SRPC and CPC:

$$sr = \begin{pmatrix} \text{false} \\ \text{false} \\ \text{true} \\ \text{true} \\ \text{false} \end{pmatrix} \quad ; \quad p = \begin{pmatrix} \text{false} \\ \text{true} \\ \text{false} \\ \text{false} \\ \text{true} \\ \text{false} \\ \text{false} \end{pmatrix}$$

The value of sr lets us conclude that communication was performed using NEW and GOOD messages, and that the STOP and BAD messages have not been used. Array p for CPC corresponds to the case where there is no delay. In this case we may conclude that no communication paths related to STOP and BAD have been executed and that no paths involving a communications delay - including paths of messages NEW and GOOD messages - have been considered.

The values of sr and p let us conclude that additional test cases are needed for both CPC and SRPC.

6. TEST-CASE GENERATION

The test coverage analysis may lead to the identification of communications that remain untested - as presented in the preceding section. Therefore the results of test coverage analysis are used to define a goal for test case generation. For instance, the generated test-cases shall cover the sr-pairs that were not executed by the given test-cases.

In order to generate a test case the test model has to be supplemented with a TESTER component, which provides the necessary degree of freedom. The TESTER has to be able to send every possible input to the model and to accept all possible responses. In Figure 10 the TESTER component for the test model is presented.

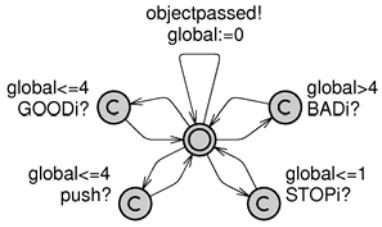

Figure 10. Test model component "Tester"

Given such a test model we can easily define reachability properties in CTL that demand the execution of one or more of the remaining sr-pairs. The reachability properties for the missing test-cases for messages STOP, GOOD, and BAD can be defined as follows:

SRPC

STOP: E<> sr[0], E<> sr[1]

BAD: E<> sr[3]

CPC

STOP: E<> sr[0]&&p[0], E<> sr[1] && p[0]

GOOD: E<> sr[2] && p[4]

BAD: E<> sr[3] && p[5], E<> sr[3] && p[6]

The model checker UPPAAL produces a shortest or fastest trace that executes the required messages. The resulting test-cases are presented in Figures 11-15. Note that we can define a property that leads to a trace that contains multiple uncovered sr pairs. The use of reachability properties for single sr-pairs/-paths can lead to decreased execution time and memory consumption. If instead we use properties that represent covering several elements of a model we get fewer test-cases but these have a relatively high cost. In addition, there may be no test-case that covers a given combination of elements even if each can be covered separately.

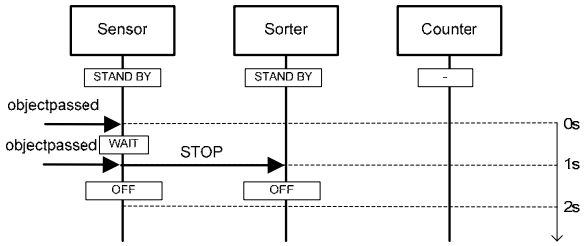

Figure 11. CPC Test Cases for STOP Message for configuration (Sensor.WAIT,Sorter.STAND_BY)

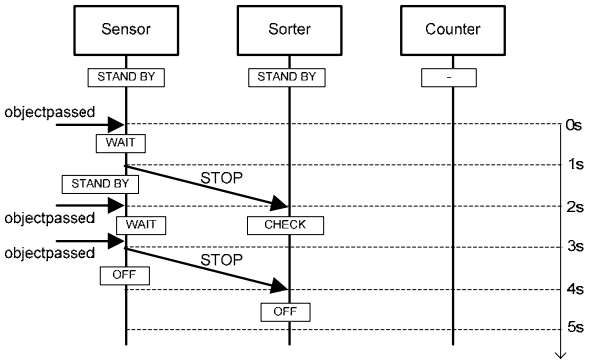

Figure 12. CPC Test Cases for STOP Message for configuration (Sensor.WAIT,Sorter.CHECK)

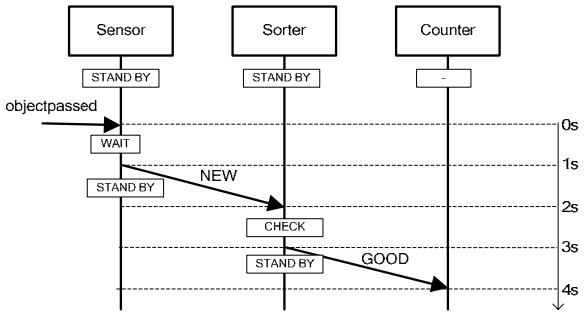

Figure 13. CPC Test Cases for GOOD Message with delay

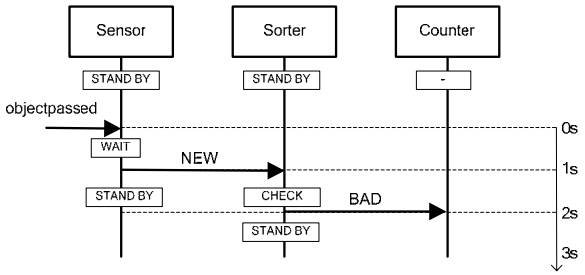

Figure 14. CPC Test Cases for BAD Message without delay

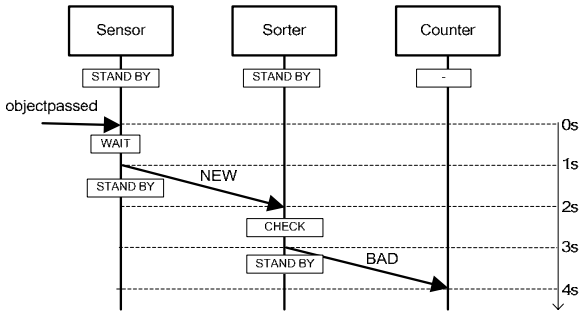

Figure 15. CPC Test Cases for BAD Message with delay

7. COMPLEXITY

The complexity of CTL model checking has been discussed in a number of publications, e.g. [9]. Here, we will briefly discuss the complexity of generating test-cases with complete sr-coverage, under the assumption that each sr-pair is feasible. Furthermore, we assume that UPPAAL is used, *breadth first search* is performed, and no complexity reduction methods are applied.

The process of checking the coverage of a test sequence is polynomial in the length of the sequence and thus we concentrate on test-case generation.

We consider a set of sr-pairs Z to be given. The length of a minimal test-sequence t that executes sr-pair $z \in Z$ is $|t|$. It is well-known that a lower bound for time consumption using CTL model checking and breadth first search is exponential in the length of the solution, polynomial in the degree of freedom of the system under test, and polynomial in the length of the CTL formula $|f|$. An upper bound for time consumption is polynomial in the overall size of the state-space and $|f|$. State space size is polynomial in the number of local states, and the size of clock and variable domains. Furthermore, it is exponential in the number of parallel automata, clocks and variables.

The length of the CTL formula for the execution of z is 1. The presented test generation method requires an additional variable for each send and receive statement and one for each send and receive pair.

Therefore, the time upper bound of our approach is exponential in the number of sr-pairs. However, the knowledge on the upper bound time consumption is only of restricted practical use as for most applications any model checker will fail to traverse the complete state space. Therefore, in practice we restrict the upper bound of test sequence lengths to a computable constant c.

8. RELATED WORK

The development of automata-theoretic testing methods was originally motivated by checking problems of sequential circuits [22]. The adoption of these methods to software has been an important research topic over decades. A detailed overview of testing methods for sequential finite state machines can be found in a number of articles, e.g. [6].

Communication coverage has been discussed in only a few recent publications. Liu and Dasiewicz presented coverage criteria on event flows in UML diagrams and proposed the use of model checkers for coverage analysis and test-case generation in [21]. Their definitions of related pairs and related paths correspond to sr-pairs and sr-paths in this paper. Here, we extend this approach to real-time systems with data extensions and solve the path feasibility problems described in [21]. Furthermore, we generalize the approach through the use of interface automata and CTL for system and requirements specification, thus our results also apply to UML specifications as used in [21].

In [2] Aizenbud-Reshef presents an abstract approach to message flow modeling and analysis. A message flow is defined as a directed graph consisting of message processing nodes and message flow connections. Coverage criteria are defined on the elements of a message flow graph. An execution of a flow model follows the connections in the graph from a message source to its target point. We can easily relate our results to the approach of Aizenbud-Reshef when we assume abstract communication specifications, e.g. sequence diagrams, as a message flow model and the system description in the form of interface automata to be the execution model. Under these assumptions SRPC subsumes Connection Coverage and CPC subsumes Path Coverage as used in [2].

Testing of distributed systems on the basis of state-based specifications using CTL model checking, and the model checker UPPAAL, was addressed by Cardell-Oliver in [7]. However, only a brief description of test selection and test completeness is given; here, we present coverage measures that can be used to address these issues.

In [19], Khoumsi presents a formal description of test execution on a timed and distributed system. In this paper controllability and observability are defined for these kinds of systems. Informally, controllability is the capability of the testing system to force the software under test to receive inputs in a given order. In our approach, we are using a centralized testing architecture and a reliable communication system [19]. We generate test cases with respect to the timing and data restrictions of the system under test. Therefore, the resultant test-cases are always feasible and there are no controllability problems, but there need not exist a test-case that satisfies a specific testing goal. Observability is the capability of the testing system to observe the outputs of the software under test and to determine the input that was the cause of an output. Again, we rely on the centralized test architecture and the reliability of our communication system, which allows us to observe input/output pairs within a constant timing delay. The approaches of Cardell-Oliver and Khoumsi deal with deterministic closed systems. By contrast, in this work we consider both closed and open systems.

A number of approaches that use model checkers for test-case coverage analysis and generation have been proposed [15][16][17]. In [26], [27], or [28] we adapted these approaches to component black box testing, also referred to as protocol testing [6], which allows the tester to deduce the intra-component coverage through observation of the interface input/output behavior only. During inter-component testing it is often important to reduce the effect of testing on the system behavior. Similarly to [26], [27], and [28] we used CTL for the expression of coverage criteria and complemented the component's behavior specification. Therefore it should be relatively straight-forward to combine both approaches in order to achieve complete test coverage at the inter- and intra-component levels.

9. CONCLUSION

Testing of real-time systems consisting of several parallel components frequently turns out a non-trivial task. The choice of test cases and reasoning about test quality on the basis of structural coverage criteria is insufficient, if criteria intended for single components only were applied, e.g. transition coverage.

In this paper we have proposed a set of new coverage criteria that aim at the inter-component level and complement coverage criteria for the intra-component level.

The proposed SRPC and CPC criteria measure coverage on the basis of message send and receive statements and inter-components paths. There exists no relation to criteria that measure coverage on the intra-component level, e.g. state coverage or transition coverage. Therefore it is necessary combining both approaches, which can be done by complementing CTL formulas and component specifications as presented in [26], [27], or [28].

The presented approach relates to the following aspects of software quality assurance: 1) Faults in the system design may be revealed during test-case generation, when the desired coverage can not be achieved due to unintended infeasible communication relations. 2) Faults in the test design may be revealed, when test cases do not achieve the desired coverage. 3) It is possible to generate test suites that achieve the desired coverage. A potential

additional advantage of this approach is that it provides a visual description of the state based testing of distributed systems, which may be beneficial in other contexts such as education and program comprehension.

Our approach is inspired by data-flow testing, specifically when applied to object-oriented programs as proposed in [12]. The annotation of def-use pairs with probabilities of executions, devised for testing at the inter-class level when there is dynamic binding, may be applied analogously to the inter-component level when dealing with uncertain SR-pairs and unreliable communication environments. The use of probabilistic annotations in combination with statistical testing [24] seems promising and might allow our approach to be extended towards non-functional testing at the inter-component level. A corresponding method for synthesizing a Markov chain from test-cases generated using the communication coverage criterion is currently under development and validation.

There is evidence that the coverage and fault detection ability of a test suite are related and that different forms of coverage complement [18]. However, recent work has shown that test sequences generated to provide state, transition and decision coverage of a specification need not be effective [14]. There is thus a need for experimental work to investigate the effectiveness of test suites produced to provide communication coverage and whether communication coverage complements other forms of coverage.

The problem of generating a test-case from parallel state-machines is reduced to the definition of a test model and properties in CTL that correspond to the coverage of some aspect of communication. The generation of the test-cases is automatically achieved by a model checker. For this reason the complexity and the capability of our approach depends mainly on the model checker used. This includes the generation of (time) optimal results, if we want either the shortest or fastest traces.

While a particular model checker, UPPAAL, was used, it should be relatively straightforward to adapt the approach for use with other model checkers. In order to improve the 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 [9], has not been extensively explored but will form part of our future work.

Promising results were obtained when the approach was applied to an experimental drive-by-wire system of a radio controlled car, which is developed as a demonstrator at Fraunhofer IESE. We intend to carry out a number of industrial and experimental case studies in order to investigate properties such as the actual mean test sequence length produced. We also plan to apply the presented approach to systems where global data supplements inter-component interaction. Furthermore, we will concentrate on an extension to the intra-component level in order to ease the location of faults in the component implementation.

10. REFERENCES

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

[2] Aizenbud-Reshef N., *Coverage Analysis for Message Flows*, 12th International Symposium on Software Reliability Engineering (ISSRE 2001), pp. 276-286. IEEE 2001,

[3] Alfaro L. de, Henzinger T. A., *Interface Automata*. Proc. of the 8th European Software Engineering Conference (ESEC 2001). pp. 109-120. ACM. 2001.

[4] Alfaro L. de, Henzinger T. A., Stroelinga M., *Timed Interfaces*. Proc. of the Second International Conference on Embedded Software, (EMSOFT 2002), 2491 pp. 108-122. LNCS. Springer, 2002.

[5] 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.

[6] Bochmann G. v., Petrenko A., *Protocol Testing: Review of Methods and Relevance for Software Testing*. Proc. of the International Symposium on Software Testing and Analysis (ISSTA 1994), pp. 109–124. ACM Press. 1994.

[7] Cardell-Oliver R., *Conformance test experiments for distributed real-time systems*. Proc. of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), pp. 159-163. ACM 2002.

[8] Chow T.S., *Testing Software Design Modeled by Finite-State Machines*. IEEE Transactions on Software Engineering, 4(3) pp. 178-187. 1978.

[9] Clarke E. M., Grumberg O., Peled D. A., *Model Checking*. MIT Press. Boston. 2000.

[10] Clarke E.M., Emerson E.A., *Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic*. Proc. of the Workshop on Logic of Programs. Yorktown Heights, NY, LNCS 131 pp. 52-71., Springer Press. 1981.

[11] Gill A., *Introduction to the Theory of Finite-State Machines*. McGraw Hill. Berkeley. 1962.

[12] Harrold M. J., Rothermel G., Performing dataflow testing on classes, Proc. Symposium Foundations of Software Engineering, ACM, 1994.

[13] Hennie F.C., *Fault-Detecting Experiments for Sequential Circuits*. Proc. of the Symposium on Switching Circuit Theory and Logical Design NJ, pages 95-110. 1964.

[14] Heimdahl M., George D., Weber R., *Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria?* Proc. 8th Intern. Symp. High Assurance Systems Engineering (HASE'04), Tampa, Fl, IEEE, 2004.

[15] Hong H.S., Cha S.D., Lee I., Sokolsky O., Ural H., *Data Flow Testing as Model Checking*, Proc. of International Conference on Software Engineering (ICSE '03), pp. 232--242, May 2003.

[16] Hong H., Lee I., Sokolsky O., Ural H., *A Temporal Logic Based Theory of Test Coverage and Generation*, International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS2002), April 8-11, 2002.

[17] Huhn M., Mücke T., *Generation of Optimized Test suites for UML Statecharts with Time*. Testing of Communicating Systems (TestCom'04). Oxford. Springer. 2004.

[18] Hutchins M., Foster H., Goradia T., Ostrand T. J., *Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria*. Proc. of Intern. Conf. on Software Engineering, Sorento, Italy, ACM, 1994.

[19] Khoumsi A., *A Temporal Approach for Testing Distributed Systems*. IEEE Trans. on Software Engineering, 28(11) pp 1085-1103, Nov 2002.

[20] Lee D., Yannakakis M., *Testing Finite-State Machines: State Identification and Verification*. IEEE Transactions on Computers, 43(3): 306-320. 1994.

[21] Liu W., Dasiewicz P., *Component Interaction Testing Using Model Checking*, Canadian Conference on Electrical and Computer Engineering, 1 pp. 41 - 46, IEEE, 2001.

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

[23] Offutt A. J., Xiong Y., Liu S., *Criteria for Generating Specification-based Tests*, Proc. 5th Intern. Conf. on Engineering of Complex Computer Systems, ACM, 1999

[24] Prowell S. J., Poore J. H., *Computing system reliability using Markov chain usage models*, Journal of Systems and Software, 73(2) pp. 219-225, Elsevier, 2004.

[25] Queille J.P., Sifakis J., *Specification and Verification of Concurrent Systems in CESAR*, Proc. 5th Intern. Symp. on Programming, pp. 337-351. 1982.

[26] 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.

[27] Robinson-Mallett C., Liggesmeyer P., *State Identification and Verification using a Model Checker*. Proceedings of the Software Engineering 2006. GI Edition Lecture Notes in Informatics, Leipzig, March 2006

[28] Robinson-Mallett C., Mücke T., Liggesmeyer P., Goltz U., *Extended State Identification and Verification using a Model Checker*. Journal on Information and Software Technology, 48 (10) pp. 981-992. Elsevier. October 2006