Using Communication Coverage Criteria and Partial Model Generation to assist Software Integration Testing

Christopher Robinson-Mallett
Berner & Mattner Systemtechnik GmbH
Berlin, Germany
robinson-mallett @berner-mattner.com

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

Jesse Poore
University of Tennessee
Knoxville, USA
poore @cs.utk.edu

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

ABSTRACT

This paper considers the problem of integration testing the components of a timed distributed software system. We assume that communication between the components is specified using timed interface automata and use computational tree logic (CTL) to define communication-based coverage criteria that refer to send- and receive-statements and communication paths. The proposed method enables testers to focus during component integration on such parts of the specification, e.g. behaviour specifications or Markovian usage models, that are involved in the communication between components to be integrated. A more specific application area of this approach is the integration of test-models, e.g. a transmission gear can be tested based on separated models for the driver behaviour, the engine condition, and the mechanical and hydraulical transmission states. 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 or generate test sequences that achieve the goal. Given the generated test sequences we derive a partial test-model of the components from which the test sequences are derived. The partial model can be used to drive further testing and can also be used as the basis for producing additional partial models in incremental integration testing. While the process of deriving the test sequences could suffer from a combinatorial explosion, the effort required to generate the partial model is polynomial in the number of test sequences and their length. Thus, where it is not feasible to produce test sequences that achieve a given type of coverage it is still possible to produce a partial model on the basis of test sequences generated to achieve some other criterion. As a result, the process of generating a partial model has the potential to scale to large industrial software systems. 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. A potential additional benefit of the 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 comprehension.

Keywords

Integration 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 the integration of distributed systems consisting of independently executing components. The proposed method enables testers to focus during component integration on such parts of the specification, e.g. behaviour specifications or Markovian usage models, that are involved in the communication between components to be integrated. A more specific application area of this approach is the integration of test-models, e.g. an automatic transmission control software can be tested based on separated models for the driver behaviour, the engine condition, and the mechanical and hydraulical transmission states. When generating a test with a specific goal, e.g. testing of clutch behaviour during automatic shifting, error conditions, or shift times, it is possible that only a subset of the product of all involved test models is needed. The proposed model integration approach allows testers to choose between such model parts of interest at the price of an overall incomplete model coverage, but which is computable within relatively low time and space bounds.

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 [21]. 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, [22]). 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]).

We assume that the software under test consists of several black-box components and that we have state-based models of each component. We also assume that these models are in the form of timed interface automata [4]. In [3] the use of parallel finite automata is proposed in order to describe communication between components in a distributed system. In [4] the approach is expanded to timed specifications. De 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 timed interface automata in order to test the communication between components and in order to provide a framework for expressing integration testing problems.

This paper introduces a new approach for the integration testing of real-time distributed systems. The first step is to generate a set of test sequences that check the interaction between components, potentially by using a predefined test criterion and a model checker. These test sequences are then combined in order to form a test-model, which is a partial model of the system. A partial model is used since the generation of a complete model can encounter the state explosion problem; in the worst case the number of system states grows exponentially as the number of components increases. This second step extends the approach reported in [28] to use timed interface automata rather than interface automata and can thus form the basis of the integration testing of real-time systems. As a side-effect of our method, the partial test-model may contain transitions that are not covered by the test sequences from which it was constructed and thus it is possible to generate further test sequences that increase the coverage of the product component specifications. In addition, the approach is designed to support incremental integration testing: we can produce a partial model for two or more components and extend this when integrating additional components. In [28] we presented an approach to generate Markov-chain test-models from interface specifications for use in statistical testing [23]; the partial model described in this paper can be used in a similar way.

While the initial test sequences can be produced in any way, we describe the use of a model checker since this ensures that our approach resolves timing and feasibility problems both on the component and on the inter-component level. We use a model checker for both coverage analysis [27] and test sequence generation and use a subset of the Computational Tree Logic (CTL) [10] (see, e.g. [9], for an overview of alternative approaches to model checking). The focus is on covering the possible communications between machines, as represented by the sending and receiving of messages. In this paper we define four coverage criteria: demanding the coverage of either all send or receive statements of a machine; insisting that all possible send/receive pairs are covered; and recognizing that the messages will be sent through an underlying network and there may be different paths through this network and so insisting that for each send/receive pair we use every possible path[1] through the network. We focus on testing from deterministic state-based specifications whose semantics can be described in terms of timed interface automata [3], but it should be possible to use the approach with any state-based formalism. Since we use a centralized test architecture we avoid the controllability and observability problems described by Cardell-Oliver [7] and Khoumsi [19].

A main contribution of this paper is the description of an incremental approach to the integration testing of a concurrent real-time system supported by a partial test-model that has been formed from a set of test sequences. While the method relies on the existence of test sequences, we describe how a model checker can be used in order to generate test sequences that 'cover' the communications in the system model. Since each possible communication can occur on many different paths through the system, the partial test-model can be smaller than the complete model. The use of a model checker in the initial test sequence generation phase ensures that 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 concurrent 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 comprehension.

This article is divided into nine sections, including the introduction in Section 1. In Section 2 an example of a distributed system is presented. In Section 3 the relevant basics regarding timed interface automata are presented and Section 4 describes some forms of test coverage at the inter-component level. In Section 5 we present an integration approach based on communication coverage and an algorithm for generating timed test-models that can support incremental integration testing. In Section 6 we present an industrial application of the presented approach. 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.

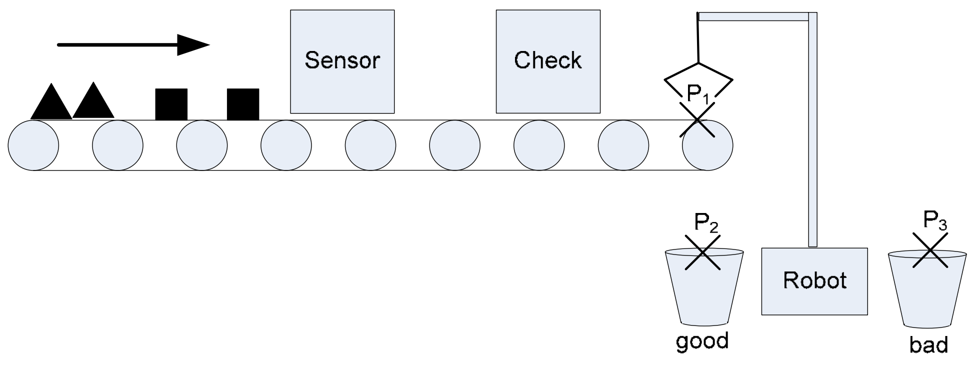


Figure 1. Example "Sorting Line" of a Distributed System

[1] where infinitely many paths exists, we recommend the use of appropriate path coverage criteria, e.g. Boundary-Interior Coverage

2. EXAMPLE

An example of a sorting line is presented in Figure 1. The distributed system consists of three software controlled components. On a conveyer belt objects, here triangles and squares, pass a sensor. The sensor signals after 1 second that an object has passed, so that this object will pass the sorter in another 1 second. The sorter checks the object and signals to the Robot whether the next object that is expected at P1 in 1 second is good or bad. The Robot puts the good objects into the left basket (P2) and the bad objects into the right basket (P3). Within a second the Robot puts an object in either basket P2 or P3, and returns to P1. Having returned to P1, the Robot signals that the job is done and another object may be processed. If an object enters the sensor while an object is being processed, the whole system immediately stops. We assume that the underlying reliable communication system provides an immediate delivery of messages. Test sequence generation using communication systems with delay are considered in [27].

In 1995 the model checker UPPAAL was presented [5]. It supports an extended version of timed automata [1]; such a timed automaton can be produced from a timed interface automaton. Some of the extensions are integer variables and constants, send (!) and receive (?) synchronization, broadcast channels, and committed locations. Synchronization over a channel e is defined between a sending transition (e!) and a receiving transition (e?). A broadcast channel allows synchronization of multiple automata in one step. In a committed location time must not pass and an outgoing transition must be taken immediately.

The example in Figure 2, Figure 3, and Figure 4, presents UPPAAL timed automata (UTA). The communication between the components is performed through broadcast channels OBJECT, NEW, GOOD, BAD, STOP, and DONE, thus any component may receive any message during execution. The initial locations are marked through double lines. Committed locations, i.e. such that time must not pass, are marked with the letter C. The components use continuous time clocks c1, c2, and c3, which may be initialized or reset on transitions and are used in constraints. In component Robot the integer variables gc and bc are used as counters of good and bad passed objects. Furthermore, the automata contain definitions of coverage variables s and sr whose use is described in Section 4.

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.

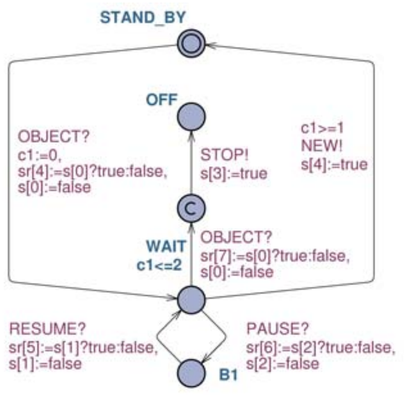


Figure 2.   UTA component "Sensor"

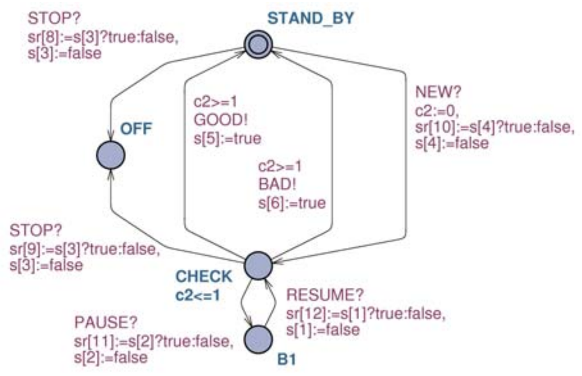


Figure 3.   UTA component "Sorter"

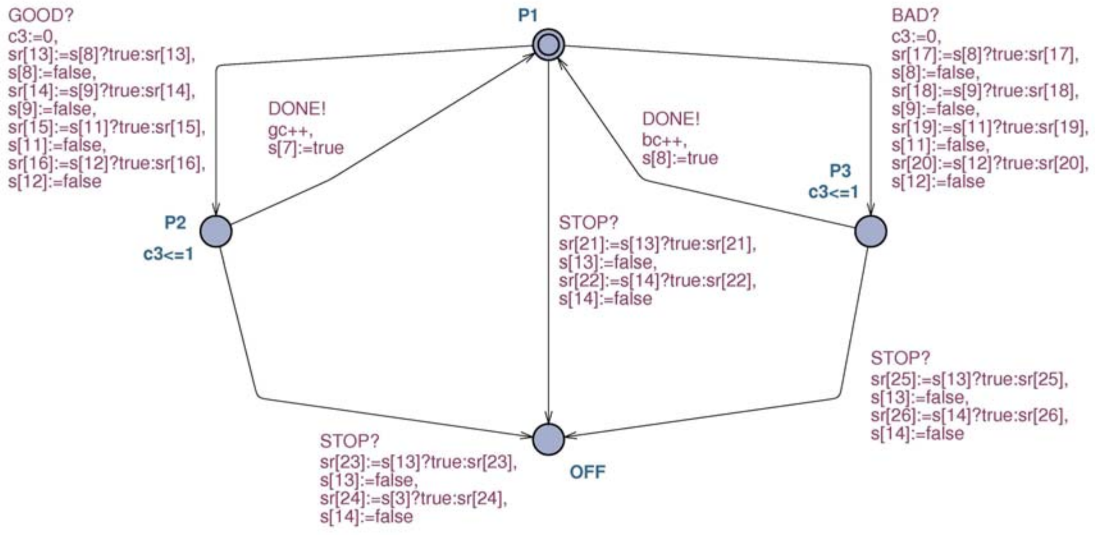

GOOD?
c3:=0,
sr[13]:=s[8]?true:sr[13],
s[8]:=false,
sr[14]:=s[9]?true:sr[14],
s[9]:=false,
sr[15]:=s[11]?true:sr[15],
s[11]:=false,
sr[16]:=s[12]?true:sr[16],
s[12]:=false
P1
BAD?
c3:=0,
sr[17]:=s[8]?true:sr[17],
s[8]:=false,
sr[18]:=s[9]?true:sr[18],
s[9]:=false,
sr[19]:=s[11]?true:sr[19],
s[11]:=false,
sr[20]:=s[12]?true:sr[20],
s[12]:=false
DONE!
gc++,
s[7]:=true
DONE!
bc++,
s[8]:=true
P3
c3<=1
P2
c3<=1
STOP?
sr[21]:=s[13]?true:sr[21],
s[13]:=false,
sr[22]:=s[14]?true:sr[22],
s[14]:=false
STOP?
sr[25]:=s[13]?true:sr[25],
s[13]:=false,
sr[26]:=s[14]?true:sr[26],
s[14]:=false
STOP?
sr[23]:=s[13]?true:sr[23],
s[13]:=false,
sr[24]:=s[3]?true:sr[24],
s[14]:=false
OFF

Figure 4.   UTA component "Robot"

The example will be used throughout the paper in order to demonstrate the proposed integration procedure. Briefly, for this example the approach follows the following steps, which are outlined in detail in Section 5:

A. Definition of a set of test sequences in the form of communication paths between two components "Sensor" and "Sorter". The example 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. Test sequence generation is hard to perform manually once models exceed a certain complexity. In [27] and in Section 5.1 we discuss how a tests suite can be produced for the example under different assumption regarding the communication system.

B. Generation of a partial model, called Sensor⊕Sorter, from the test sequences. The state space of Sensor⊕Sorter is a subset of that of the model Sensor⊗Sorter formed by composing Sensor and Sorter. In Section 5.2 we present an approach to derive a partial model from a finite set of test sequences. The partial model Sensor⊕Sorter can be used as the basis of further testing of the subsystem formed by integrating the implementations of Sensor and Sorter.

C. Once we have finished testing the subsystem formed from the Sensor and Sorter components, we extend the partial model Sensor⊕Sorter to incorporate "Robot" and so form Sensor⊕Sorter⊕Robot. The resultant partial model is used to test the final system composed of the Sensor, Sorter and Robot components.

Note that this approach can be used for incremental integration testing since the procedure is identical for the cases where there are two components, there are two composite models, or there is a composite model and a component.

3. TIMED INTERFACE AUTOMATA

In this section we describe timed interface automata, which are used in this paper because of their ability to describe the visible behavior of a distributed system in a straightforward and generic fashion. The definition of timed interface automata is taken from the work of de Alfaro and Henzinger [4], which we recommend for a detailed study. A timed interface automaton is defined by a tuple $P=(V_P,V_P^{init},X_P,A_P^I,A_P^O,Inv_P^I,Inv_P^O,R_P)$, where V_p is a finite set of states, $V_P^{init}\subseteq V_P$ is a set of initial states, and X_P is a finite set of clocks. A_P^I and A_P^O are mutually disjoint sets of input and output actions and $A_P=A_P^I\cup A_P^O$. The set A_P is disjoint from the set T of timed actions that simply represent the passing of time. For a state v, $Inv_P^I(v)$ and $Inv_P^O(v)$ are the input and output invariants on the clocks for state v; these give conditions under which a state may be left or entered. Thus, a transition can only leave a state if the state’s invariant holds and a transition can only enter a state if that state’s invariant holds; whether we consider the input or output invariant depends on whether the action that triggers the transition is an input or an output. $R_P\subseteq V_P\times C(X_P)\times A_P\times P(X_P)\times V_P$ is the transition relation; here $C(X_P)$ denotes the set of guards on clocks and for a set B, P(B) denotes the powerset of B. A transition (v,g,a,X’,v’) should be interpreted in the following way: if P is in state v and guard g on the clocks holds then action a can lead to P moving to state v’ and the clocks in X’ being reset but this can only happen if the invariant for states v holds for the clock valuations before the transition and the invariant for v’ holds after the reset of the clocks in X’ (if a is an input then the input invariants must hold and otherwise the output invariants must hold).

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

The composition of timed 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 timed 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 [4] for a detailed definition. Since P and Q are not necessarily input enabled [4], some transitions present in P or Q may not be present in P⊗Q; in P⊗Q we may reach a state where the current state of P suggests that it can follow a transition but the current state of Q does not allow this.

4. COVERAGE CRITERIA FOR PARALLEL TIMED AUTOMATA

Typical coverage criteria for sequential specifications, such as state coverage or transition coverage, are defined for the case where we have only one automaton. When we have many parallel machines P_1, …, P_m we could apply these criteria to the product automaton $P_1 \otimes \ldots \otimes P_m$ formed by composing $P_1, \ldots, P_m$ but this can lead to a state explosion problem. Alternatively, we could apply these criteria to the individual machines $P_1, \ldots, P_m$ but this does not consider how the automata can interact.

This section describes test criteria that consider the ways in which the P_i can interact but do not require the entire product automaton $P_1 \otimes \ldots \otimes P_m$ to be formed. Test criteria for the coverage of communication in a model formed using (untimed) interface automata have been described [27]. Here we extend these test criteria to timed interface automata. These criteria do not require us to have the product automaton at hand; we can use a model checker to produce parts of this during test sequence generation. We know that each send and receive pair in the P_i 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.

Definition

sender coverage: In order to achieve complete sender coverage every send statement in each P_i has to be executed at least once.

Definition

receiver coverage: In order to achieve complete receiver coverage every receive statement in each P_i has to be executed at least once.

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 complete 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 an underlying 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 for every sr-pair.

Clearly complete sr-paths coverage subsumes complete sr-pairs coverage. Since each sr-pair may occur in many paths of $P_1 \otimes \ldots \otimes P_m$, it may not be necessary to explore the entire state space of the product automaton $P_1 \otimes \ldots \otimes P_m$ in order generate a set of test sequences that provide complete sr-pairs coverage or complete sr-paths coverage.

In addition to sr-pairs coverage we might also demand transition coverage. Transition coverage applies to parallel automata as described in [17]; therefore we recommend this work for a detailed description.

5. INTEGRATION TESTING PROCEDURE

Given state-based models $P_1, \ldots, P_m$ of the components, integration testing can be based on the product automaton $P_1 \otimes \ldots \otimes P_m$ but the process of generating $P_1 \otimes \ldots \otimes P_m$ can suffer from the state-explosion problem. Instead, we propose the use of partial models, called *test-models*, which are generated from the test sequences. These test sequences might have been created in a manual or automatic manner during component testing.

Let us suppose that we have tested a subsystem formed from components that are intended to implement P_1, …P_k using a test-model $M=P_1\oplus\ldots\oplus P_k$ and are now going to introduce a new component P_{k+1}. We can now generate test sequences from M and P_{k+1}. The resultant test sequences can be used in order to form a new test-model $M'= P_1\oplus\ldots\oplus P_{k+1}$, which can now be used in testing the new subsystem. This process can be repeated until all of the components have been integrated. Naturally, even if we use a criterion such as all sr-pairs, this iterative process could lead us to failing to cover some feasible sr-pairs as a result of the test sequences chosen in devising an earlier test-model. Where this occurs, it is possible to extend a test-model with additional test sequences. Furthermore, the tester component, e.g. such as presented in Figure 6, can be used to predefine the interface of M', either by creating or by demanding specific coverage of senders in the tester component so that M' will possess the desired open receivers.

At each integration step, the size of the partial model is linear in the total number of steps included by the chosen test sequences. Of course, the complexity of integrating a system depends on the procedures used for generating sequences from models and generating models from sequences.

In Section 5 we will use the sorting line example to demonstrate the proposed integration strategy. In a first step we will demonstrate the integration of Sensor and Sorter and the generation of a test-model based on test sequences produced to satisfy a coverage criterion. In the second step component "Robot" is integrated making use of the test-model generated during the integration of Sensor and Sorter.

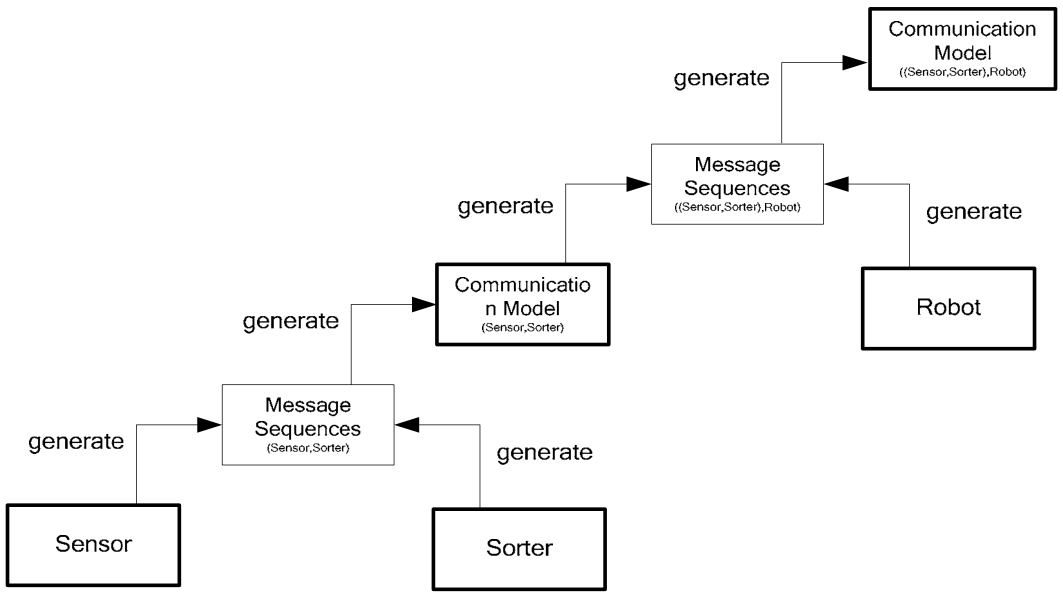


Figure 5. Integration process for the sorting line example

5.1 Test Sequence Generation

A test sequence t is represented by a sequence $v_0,a_0,v_1,a_1,\ldots,v_n$ in which $v_0,\ldots,v_n$ are states of the product $P=P_1\otimes\ldots\otimes P_m$, $a_0,\ldots,a_{n-1}$ are actions of P, and $(v_0,q_0,v_1),\ldots,(v_{n-1},a_{n-1},v_n)$ are transitions of P. We denote the states of component P_i using terms of the form v_{ij}. A state of P is represented by a tuple $v=(v_{1i},\ldots,v_{mj})$. The initial state of P is $v_0=(v_{10},\ldots,v_{m0})$.

For example, the test sequence in Fig. 7 is represented by the sequence v_0,a_0,v_1,a_1,v_2 in which:

- v_0 is the initial state (Sensor_STAND_BY,Sorter_STAND_BY),
- v_1 is the state (Sensor_WAIT,Sorter_STAND_BY),
- v_2 is the state (Sensor_OFF,Sorter_OFF), and
- a_0 is the action OBJECT? and a_1 is the action OBJECT?,STOP!.

The test sequence t represented by v_0,a_0,v_1,a_1,…,v_n defines a state structure that has state set V(t)={v_0,…,v_n} and we let R(t) denote the set of transitions of P that are in t. Note that while each transition in t is a transition from the product automaton, it is not necessary to generate the product automaton in order to find the set R(t)={(v_0,q_0,v_1),...,(v_{n-1},a_{n-1},v_n)} of transitions in t. For example, in the above sequence we have two transitions: one from state (Sensor_STAND_BY,Sorter_STAND_BY) to state (Sensor_ WAIT,Sorter_STAND_BY) with action OBJECT? and one from state (Sensor_ WAIT,Sorter_STAND_BY) to state (Sensor_OFF,Sorter_OFF) with action OBJECT?,STOP!. Note that in this paper we only consider deterministic models; extending the approach to allow non-determinism is a topic for future work.

Here we use a model checker for test sequence generation based on sr-pair coverage. Of course, other coverage criteria can be used. The effect of different coverage criteria on the quality of integration test-sequences is an important area of future research.

The coverage of sr-pairs is measured by adding variables to each model, which might be either a component specification P_i or a test model as described in Section 5.2. For each send statement one Boolean global variable s is needed, which is set true once the send statement has been executed. For each send and receive pair a Boolean global variable sr is introduced and this is set to true directly after the execution of the corresponding pair, i.e. once a receive statement is executed and a corresponding send variable s is true. After updating the sr variable the value of the corresponding s variable is reset to false. The UTA test-models in Figure 2, Figure 3, and Figure 4 include variables s and sr in order to record the sr-pairs executed. For example, in Figure 2 for "Sensor" we see that the STOP! statement is followed by the definition s[3]:=true and corresponding pair variable sr[8] and sr[9] are defined in Figure 3 "Sorter".

In order to generate a test sequence the test-model is augmented with a "Tester" component, which is able to send every possible input to the model and to accept all possible responses. In Figure 6 the "Tester" component for the test-model is presented.

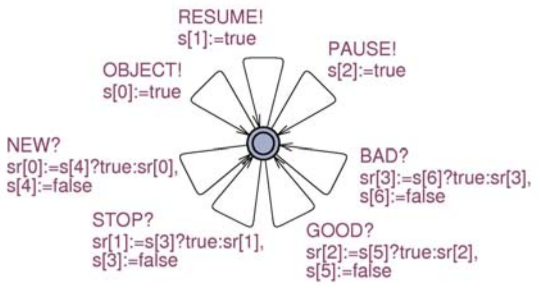


Figure 6. "Tester" component

Given a test-model we can define reachability properties in CTL that demand the execution of one or more of the sr-pairs that have not been covered. For instance, to generate a test sequence that sends the NEW message from "Sensor" to "Sorter" we use the CTL formula E<> sr[10]. For more details regarding communication based test sequence generation we refer to [27].

The model checker UPPAAL produces a shortest or fastest trace that executes the required messages. Note that we can define a property that leads to a trace that covers multiple 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 sequences but these can be longer and their generation may take more time and memory. In addition, there may be no test sequence that covers a given combination of elements even if each can be covered separately.

5.2 Integration Model Generation

Let us suppose that we wish to produce a test-model to assist in testing a subsystem formed by integrating components that implement $P_1,\ldots,P_k$. Unfortunately, the generation of the product $P=P_1\otimes\ldots\otimes P_k$ can lead to a combinatorial explosion. For this reason we describe the construction of a test-model $P'=P_1\oplus\ldots\oplus P_k$ on the basis of a given set of test sequences. Here we describe this process for two models in order to simplify the exposition but this easily generalizes to k models.

Given two timed interface automata P_i and P_j, $P_i\oplus P_j = (V'_P, V'^{init}_P, X_P, A'^I_P, A'^O_P, Inv'^I_P, Inv'^O_P, R'_P)$ is produced using the following steps:

1. Generate test sequences $t_1,\ldots,t_l$ from P_i and P_j, possibly using a communication coverage criterion such as sr-path coverage and breadth first search.
2. Set V'_P to be the union of the sets of states contained in the test sequences. Thus, $V'_P=V(t_1)\cup\ldots\cup V(t_l)$ and is a subset of the set of states of the product automaton P. Each element of V'_P is in the form (v_i,v_j) for state v_i of P_i and v_j of P_j. For each $v\in V'_P$ we have that $Inv'^I_P(v)=Inv^I_P(v)$ and $Inv'^O_P(v)=Inv^O_P(v)$.
3. The initial state V'^{init}_P of P' is the tuple (v_{i0},v_{j0}) containing the initial state of each component model.
4. Define input alphabet A'^I_P to be the set of inputs that appear in the test sequences.
5. Define A'^O_P to be the set of outputs that appear in the test sequences.
6. Set R'_P to be the set of transitions contained in the test sequences and so initially $R'_P = R(t_1)\cup\ldots\cup R(t_k)$. The initial set R'_P is complemented with transition in an additional step. Suppose that state (v_i,v_j) is in V'_P, the action a when P_i is in state v_i affects P_i only, some test sequence includes a transition t that takes P_i from state v_i to state v'_i with action a, and the transition corresponding to t can occur in state (v_i,v_j). Then we can add the corresponding transition from (v_i,v_j) to (v'_i,v_j) to R'_P and add the state (v'_i,v_j) to V'_P if this is not already in V'_P. Note that this says that we only add such a transition if the transition can take us from (v_i,v_j) to (v'_i,v_j) and this requires there being a time that satisfies the timing constraint of the transition and the appropriate invariants of these two states[2].
7. We label a state in V'_P as the final state.

[2] i.e. if a is an input then it is the input invariants of these two states and otherwise the output invariant of the states

8. Connect any states without outgoing transitions to the final state.

The test-model $P_i \oplus P_j$ is essentially a union of the test sequences. We add copies of any transition t contained in a test sequence such that t involves just one component but only do this where timing constraints and invariants can be satisfied. Note that if one or more of P_i and P_j has been produced in an earlier stage from two or more timed interface automata then the states of P_i and P_j may themselves be tuples of states of timed interface automata.

The resultant automaton represents a subset of all possible interactions between the components. The size and quality of this test-model depends on the test sequences and thus on the test sequence generation method used. It is clear that if we have test sequences $t_1,\ldots,t_l$ and each has length at most m then the test-model has at most $m \cdot l$ states.

Distributed software systems often do not possess a final state and even if there is one it might not be in V'_P. Therefore, we mark a state of V'_P as the final state of P'. This is the only mandatory manual step in the generation of a test-model.

5.3 Integration of the sorting line example

Integration testing of the sorting line example is performed in two steps. In step 1 components "Sensor" and "Sorter" are integrated. Communication coverage, e.g. sr-pairs coverage, is applied in order to generate test sequences that focus on the interaction between "Sensor" and "Sorter". The test sequences are the basis for generating a communication model, which represents parts of the possible interaction between "Sensor" and "Sorter". The "Tester" component in Figure 6 is added to the test configuration, which accepts all open outputs and produces all open inputs.

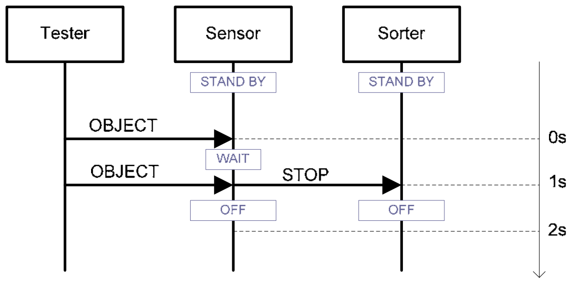


Figure 7. Test sequence to execute send-receive pair sr[8]

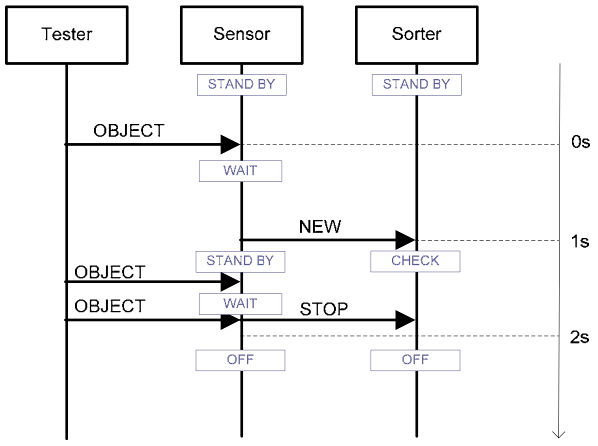


Figure 8. Test sequence to execute send-receive pair sr[9]

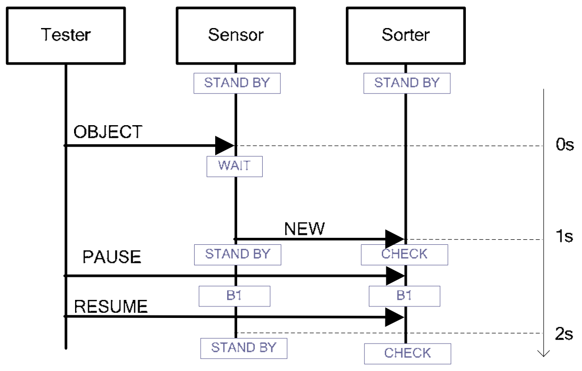


Figure 9. **Test sequence to execute send-receive pairs sr[11],sr[12]**

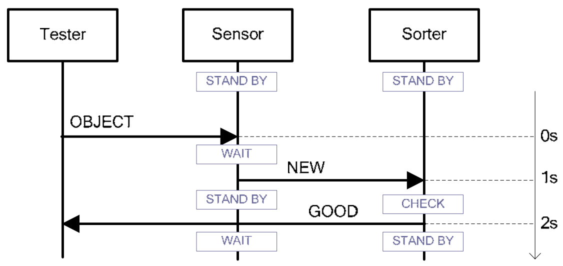


Figure 11. **Test sequence to execute send-receive pair sr[2]**

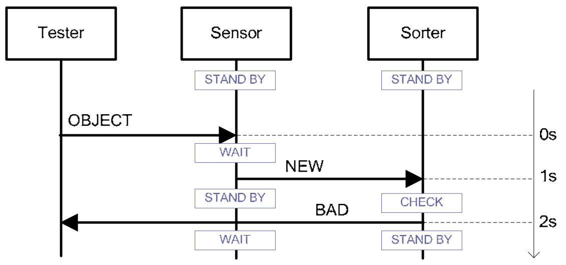


Figure 10. **Test sequence to execute send-receive pair sr[3]**

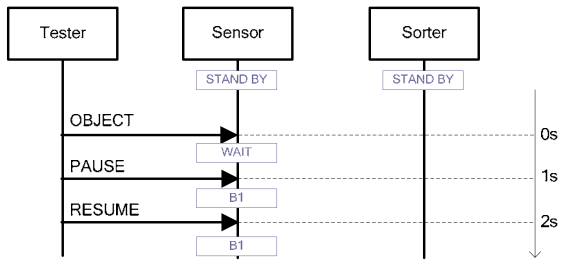


Figure 12. **Test Sequence to execute send receive pairs sr[5],sr[6]**

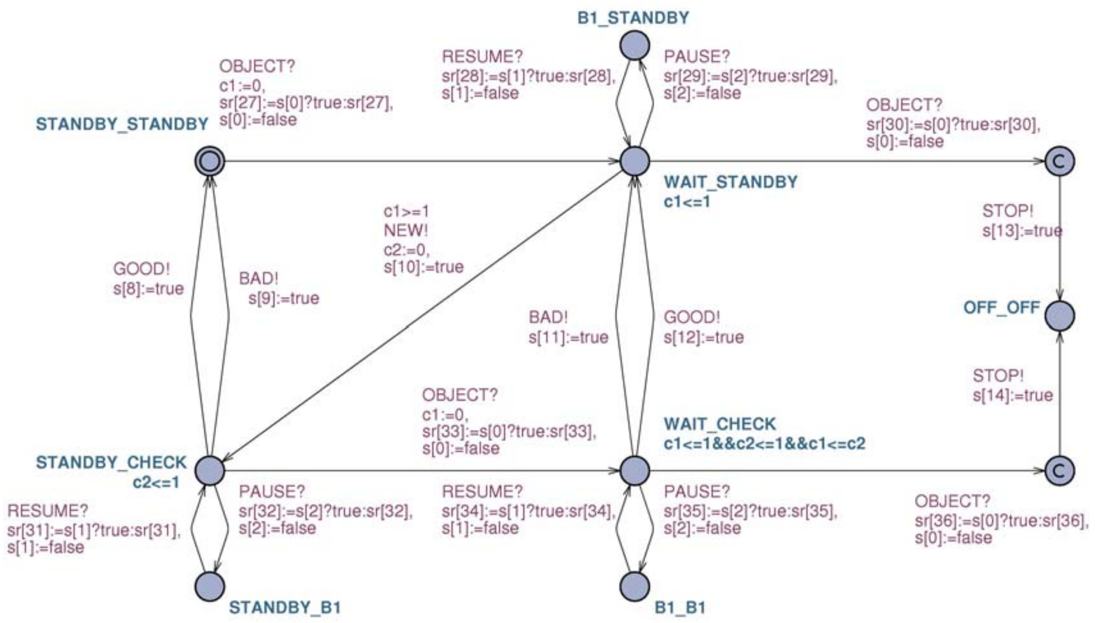


Figure 13. **Test-model (Sensor⊕Sorter) based on sr-pairs and receiver coverage**

The first step integrates "Sensor" and "Sorter" into Sensor⊕Sorter. Applying communication based test sequence generation delivers the test sequences presented in Figure 7, Figure 8, Figure 9. and Figure 10. The message DONE is not communicated between "Sensor" and "Sorter". However, we can observe at this point that this is required in the following integration step if we wish to be able to cover all sr-pairs. Therefore, we apply *receiver coverage*; which extends the test sequence in Figure 10 with the final "DONE" message from "Tester" to "Sensor". Alternatively, we integrate "Tester" into the coverage analysis and demand sender coverage, with the benefit of being able to choose the desired input signals of Sensor⊕Sorter. Currently the addition of such a test sequence is based on the tester's judgment and not the sr-pairs test criterion. This example suggests that there may be value in insisting on receiver coverage in addition to sr-pair coverage but the development of alternative test criteria and heuristics that assist later parts of the integration testing process is a topic for future work.

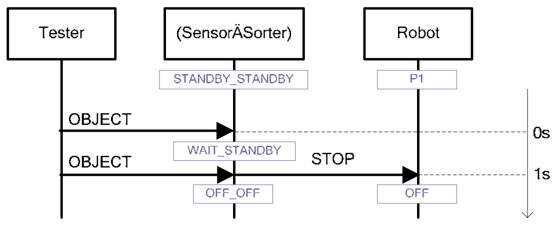


Figure 14. Test sequence to execute send-receive pair sr[21]

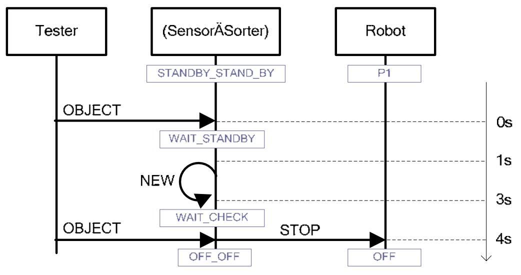


Figure 15. Test sequence to execute send-receive pair sr[22]

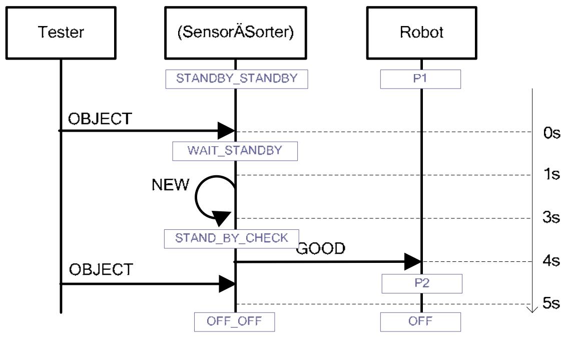


Figure 16. Test sequence to execute send-receive pairs sr[13]

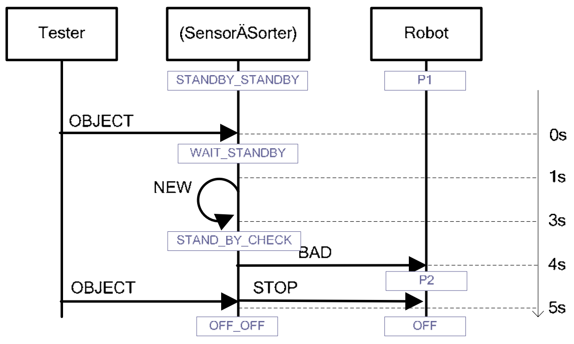


Figure 17. Test sequence to execute send-receive pairs sr[23]

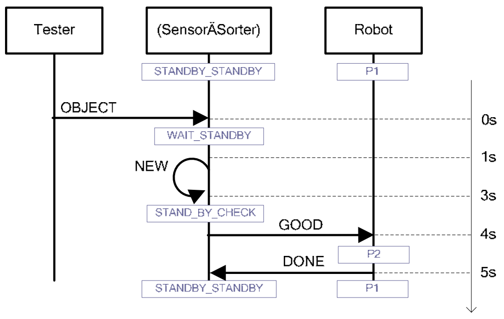


Figure 18. Test sequence to execute send-receive pair sr[20], sr[16]

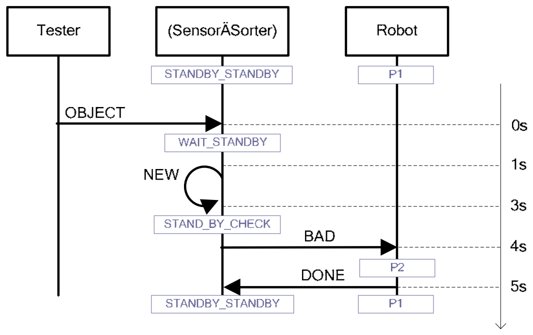


Figure 19. Test sequence to execute send-receive pair sr[21], sr[18]

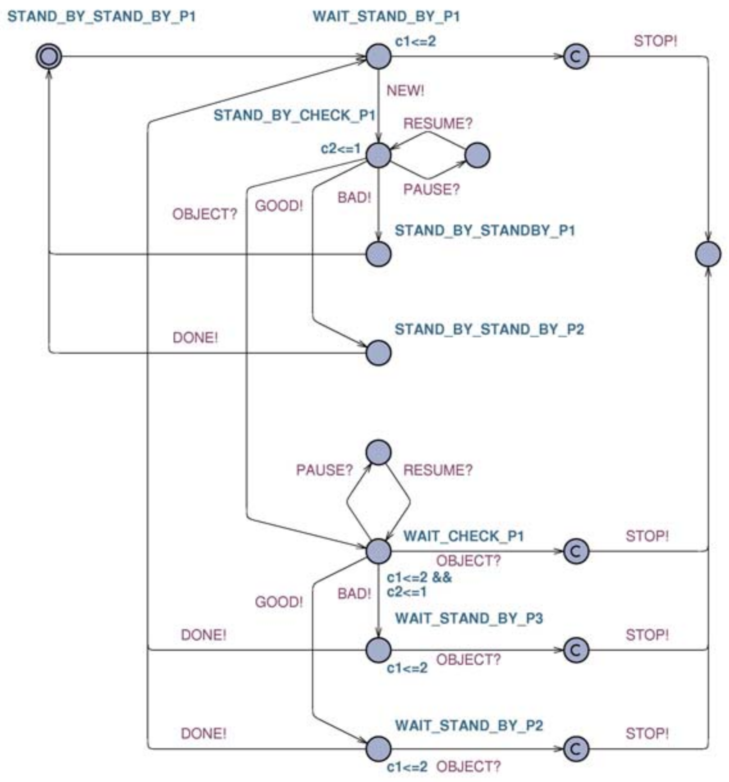
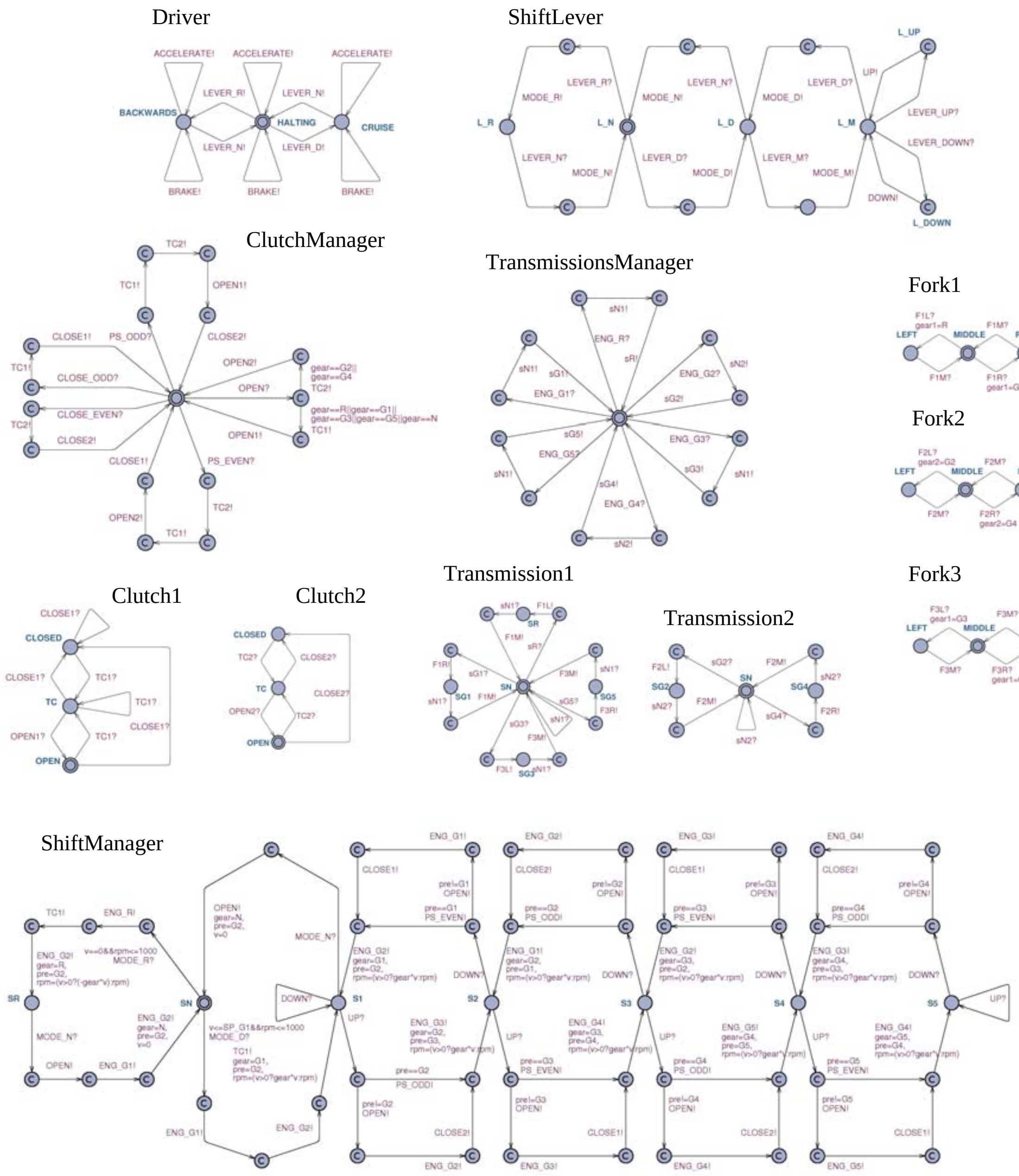
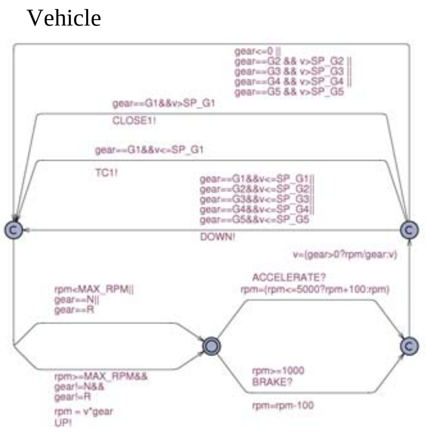


Figure 20. Test-model ((Sensor⊕Sorter) ⊕Robot)

In the second integration step we add the implementation of "Robot" to form the entire system and generate test sequences from M_1= Sensor⊕Sorter and the model of "Robot". The test sequences in Figure 14, Figure 15, Figure 16, Figure 17, Figure 18, and Figure 19 are generated from M_1 and "Robot" using sr-pairs coverage. The resulting model M_2= Sensor⊕Sorter⊕Robot is presented in Figure 20. During test sequence generation it transpired that several sr-pairs were infeasible. The final model may serve to automatically generate test sequences for the completely integrated system.

The resulting models Sensor⊕Sorter⊕Robot and Sensor⊕Sorter may be used to generate additional test sequences of arbitrary number and length, for instance for the purpose of statistical testing [23]. This is especially during testing of distributed electronic systems, e.g. automotive electronics, of great benefit, when for each sub-system statistical quality estimations can be produced. But also during integration testing of a concurrent software but not necessarily distributed software, e.g. transmission or engine controls, this approach is beneficial as test-models of different purpose and detail can be adapted to a specific integration scenario from existing specifications.





```
// Physical variables
int[-120,100] gear, gear1, gear2, pre;
int[-50,500] v=0; int[0,7500] rpm=900;
// GEAR coding
int[-2,100] N=0,G1=100,G2=60,G3=40,
            G4=30,G5=25,R=-120,U=-2;
//Downshift Speeds
int[10,95]  SP_G1=10, SP_G2=25, SP_G3=40,
            SP_G4=65, SP_G5=95;
//Upshift RPM
int[3000,3000] MAX_RPM = 3000;
```

Figure 21. Simplified Dual Clutch Transmission Control UTA

6. AN INDUSTRIAL APPLICATION

This section describes the application of the proposed integration method to the integration testing of a piece of control software for a dual clutch transmission[3].

The example presented in Figure 21 is a simplified extract of the UPPAAL model of the transmission control software. The test concept for the final validation of the transmission control software includes the automatic test-case generation for Model-in-the-Loop and Hardware-in-the-Loop testing using JUMBL [24], a JAVA library to build Markovian usage models and to generate test-cases stochastically from these. The model checker UPPAAL is used for test model development and validation.

The main simplifications of the example are as follows:

- five instead of six forward gears
- no gear skipping functions
- no safety function
- no robustness functions
- unrealistic calibration parameters
- missing technical details, e.g. no hydraulic models
- no timing constraints

The simplified transmission control UTA consists of 12 components spanning a state space of over 10^{20}, while the real transmission control software consists of more than 40 components.

Briefly, a dual clutch transmission consists of two integrated partial transmissions, one for odd and reverse gears and the other for even gears. For each transmission a separate clutch controls input torque. A transmission consists of shift forks that engage gears mechanically. In contrast to standard gear boxes, in a dual clutch transmission always two gears are engaged.

[3] GETRAG Ford Transmission GmbH (www.getrag-ford.com)

Gears are automatically shifted seemlessly by synchronously opening and closing the two clutches, if the preselected gear is the next gear in the shift strategy, e.g. during acceleration usually the next gear is expected to be the next upper gear. If gear shifts are unexpected the dual clutch behaves like a standard gear box[3]. The components of the simplified transmission control are:

- Shift Lever - Selector of shift modes for the driver
- Shift Manager - Controls shift strategy, the order of gears to engage based on shift mode, vehicle and engine speed
- Transmissions Manager - Controls the two transmissions to engage and preselect gears
- Clutch Manager - Control of the two clutches
- Clutch 1 - Opens and closes clutch 1, slipping clutch operates in torque control mode (TC)
- Clutch 2 - Opens and closes clutch 2, slipping clutch operates in torque control mode (TC)
- Fork 1 - Middle position is neutral, left is reverse gear, right is gear 1
- Fork 2 - Middle position is neutral, left is gear 3, right is gear 5
- Fork 3 - Middle position is neutral, left is gear 2, right is gear 4
- Driver - arbitrary selection of shift modes, accelerate and brake commands
- Vehicle - computation of vehicle and engine speeds on acceleration and brake commands

The dual clutch transmission model demonstrates the size and complexity of a real-world application of timed automata. Such models are used in transmission control integration testing, such that the model structure is developed and checked using UPPAAL while JUMBL is used as a test-case generator for Model-in-the-Loop and Hardware-in-the-Loop simulation. During integration testing we are often interested in only a small part of the model, when the presented approach eases slicing of the test models. Also UPPAAL is used to generate single test-cases with specific testing purpose.

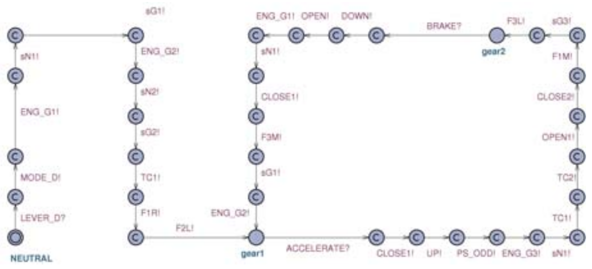


Figure 22. Test Model to execute gear 1 to gear 2 shift loops

In the agile development environment of the dual clutch transmission control software with up to two builds a week the presented approach enables testers to generate test-cases with specific purpose in the given time and allows testing of specific functionalities in the integrated software. Furthermore, as the software is tested on the target hardware observability and controllability are restricted to the hardware communication interface. Due to complexity problems while generating test-cases from the complete model, it soon became clear that a method for simplification and slicing of the complete model into computable parts was demanded that allows the generation of observable and controllable communication based test-cases.

We apply sr-coverage to UP and DOWN signals of the model presented in Figure 21 with the requirement to execute these commands until gear 1 is engaged. UPPAAL can be used to deliver traces that can be used to generate the (simplified) test model presented in Figure 22. For the ease of explanation, a simplifation of the test model is needed for ACCELERATE and BRAKE signals, of which in the real application multiple are needed to exceed the up- and downshift bounds. During transmission control verification, test models of this form are primarily used in simulation-based testing of transmission control electronics, e.g. in Hardware-in-the-Loop, and transmission control software, e.g. in Model-in-the-Loop, to generate test-cases of arbitrary length, which is required for testing the effect of long-term timers and physical parameters, e.g. temperature dependend mechanical and electronical behaviours. The test model represents the behaviour of the transmission when the driver selects driving mode and accelerates/decelerates the car so that the transmission alternates between gear 1 and gear 2. The input language TML of JUMBL allows to describe finite state machine structures, so that the generated test model can be used as an input to JUMBL. Note that using the product machine of this example as an input to JUMBL would result in a test-model with a size of over 10^{20} states, while the partial model contains only 3 states. Of course the generated test model represents only a small part of the transmission behaviour, but the proprosed model generation approach can be used to extend and combine these models where needed. Costs of generating such a partial model are in this case mainly depending on the trace length. The example trace was generated in less than a second, while the test-model is generated and verified semi-automatically using UPPAAL and JUMBL.

7. COMPLEXITY

The complexity of CTL model checking has been discussed in a number of publications, e.g. [9]. Here, we briefly discuss the complexity of generating test sequences with complete sr-pairs 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 determining which sr-pairs are covered by a test sequence is polynomial in the length of the sequence and so we concentrate on test sequence 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 test generation method presented requires the addition of a variable for each send and for each sr-pair.

The generation of a communication test-model is linear in the length and the number of test sequences. Test sequences may exist from earlier testing activities or may be constructed as the basis for the process of generating a communication test-model. In [29] we proposed a method to generate test sequences and presented a detailed discussion of the complexity of our approach and the inherent testing problem. For more details on generating test sequences from component test-models we may refer to [22] and [29].

To conclude, 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 a model checker may fail to traverse the complete state space and so, in practice we place an upper bound c on the test sequence lengths. Also, in many practical applications test sequences might be given in advance.

8. RELATED WORK

The development of automata-theoretic testing methods was originally motivated by checking problems of sequential circuits [21]. 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 sequence generation in [20]. 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 [20]. Furthermore, we generalize the approach through the use of timed interface automata and CTL for system and requirements specification, thus our results also apply to UML specifications as used in [20].

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 timed interface automata to be the execution model. Under these assumptions sr-pair coverage subsumes Connection Coverage and sr-path coverage 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 sequences with respect to the timing and data restrictions of the system under test. Therefore, the resultant test sequences are always feasible and there are no controllability problems, but there need not exist a test sequence 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 sequence coverage analysis and generation have been proposed [15][16][17], where the ability of the model-checker to produce counter-examples, in the form of traces, is extensively used. We follow the same approach, with the difference that UPPAAL produces traces into target states on demand. In [29] we adapted these approaches to component black box testing, also referred to as conformance testing [6], which allows the tester to deduce the intra-component coverage through observation of the interface input/output behavior only. Similarly to [26] and [29] 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.

In [11], Glasser et al. propose the use of state-based communication models for the testing of distributed systems and point out the need to generate partial models in order to avoid state-space explosion problems. Here, we propose a method that may be used to generate such incomplete models.

9. CONCLUSION

The testing of real-time systems consisting of several parallel components frequently turns out to be a non-trivial task. The choice of test sequences 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.

This paper describes an approach to integration testing based on partial test-models. The process is driven by a set of test sequences, possibly generated to satisfy a particular test criterion, from which the test-model is derived. The test-model can then be used as the basis for the testing of the integrated components. When additional components are to be integrated into a subsystem that has test-model M, M can be extended and so the approach can be used in incremental integration testing. While the approach presented is for components specified using timed interface automata it should be possible to adapt it to other types of state-based model. 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 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 [23] 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 sequences generated using the communication coverage criterion is presented in [28]. It should be possible to extend this approach to provide a statistical testing framework for timed distributed systems.

The size of the test-model is linear, in terms of the number and size of the test sequences. This complexity analysis assumes that the test sequences from which the test-model is generated already exist. Test sequence generation can be automatically achieved by a model checker and while a particular model checker, UPPAAL, was used, it should be relatively straightforward to adapt the approach for use with other model checkers. In particular, the use of heuristic and symbolic model checking approaches could improve performance as might complexity reduction methods such as data abstraction [9].

One possible area of future work is the use of the proposed method in optimizing configuration parameters of a distributed system or to assess non-functional quality attributes, e.g. reliability, performance, and availability. There should also be scope for applying this approach for online diagnosis, where the communication test-model may be dynamically extended during run-time. In addition, the proposed approach is designed for systems whose components are deterministic. The extension of this to non-deterministic components is a topic for future work.

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] Glasser, U.; Gurevich, Y.; Veanes, M., *Abstract communication model for distributed systems*, IEEE Trans. on Software Engineering, Volume 30, Issue 7, July 2004 Page(s):458 - 472

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

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

[22] 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.

[23] 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.

[24] Prowell S. J., *JUMBL: A Tool for Model-Based Statistical Testing*, Proceedings of the 36th Hawaii International Conference on System Sciences, IEEE, 2003

[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*. Workshop on Advances in Model-Based Software Testing (A-MOST’05). St. Louis. 2005.

[27] Robinson-Mallett C., Hierons, R. M., Liggesmeyer P., *Achieving Communication Coverage Criteria in Testing*, Workshop on Advances in Model-Based Testing 2006. Raleigh, NC. 2006.

[28] Robinson-Mallett C., Hierons, R. M., Poore J., and Bauer T., *Using Partial Models to support the Testing of Distributed Systems*, International Conference on Software Engineering and Applications (SEA 2007), IASTED, Boston, November 2007.

[29] 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.