TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Interner Bericht

Compositional Testing of Communications Systems

R. Gotzhein, F. Khendek

Technical Report 329/04

FACHBEREICH
INFORMATIK

Postfach 3049 · D-67653 Kaiserslautern

Compositional Testing of Communication Systems

R. Gotzhein[1], F. Khendek[2]

[1]Computer Science Department, University of Kaiserslautern, Kaiserslautern, Germany
gotzhein@informatik.uni-kl.de

[2]ECE Department, Concordia University, Montreal, Canada
khendek@ece.concordia.ca

Technical Report 329/04

Computer Science Department
University of Kaiserslautern
Postfach 3049
67653 Kaiserslautern
Germany

Compositional Testing of Communication Systems

R. Gotzhein[1], **F. Khendek**[2]

[1]Computer Science Department, University of Kaiserslautern, Kaiserslautern, Germany
gotzhein@informatik.uni-kl.de

[2]ECE Department, Concordia University, Montreal, Canada
khendek@ece.concordia.ca

Abstract. Today, test methods for communication protocols assume, among other things, that the protocol design is specified as a single, monolithic finite state machine (FSM). From this specification, test suites that are capable of detecting output and/or transfer faults in the protocol implementation are derived. Limited applicability of these methods is mainly because of their specific assumptions, and due to the size of the derived test suite and the resulting test effort for realistic protocols. In this work, the *compositional test method* (*C-method*), which exploits the available structure of a communication protocol, is proposed. The C-method first tests each protocol component separately for output and/or transfer faults, using one of the traditional test methods, then checks for composability, and finally tests the composite system for composition faults. To check for composability and to derive the test suite for the detection of composition faults, it is not required to construct the global state machine. Instead, all information is derived from the component state machines, which avoids a potential state explosion and lengthy test cases. Furthermore, the test suite checks for composition faults only. This substantially reduces the size of the test suite and thus the overall test effort.

1. Introduction

Systematical methods for testing protocol implementations have a long and successful record. The relevance and the potential of protocol testing is first recognized in [11], which has initiated a research stream that has produced a diversity of test methods with different foci. These methods usually assume that the design of the protocol implementation to be tested is given in the form of a finite state machine (FSM), that this state machine is minimal, completely specified, and fully connected. Some methods further assume the FSM to be deterministic [2,4,13], while others relax this constraint [9]. Recently, the focus has been shifted to real-time testing [3] and embedded testing.

All these test methods have in common that test cases are derived from a protocol design that is specified as a single finite state machine. To test a complete communication system, such a monolithic FSM has to be derived first, which normally leads to large state spaces and therefore long test cases as well as large test suites that can often not be managed in practice. Thus, complete protocol testing today is restricted to testing individual, relatively small protocol entities, but not their interworking.

It is this observation that has led us to investigate a different approach, which we call *compositional testing*. Here, communication systems are perceived as being built from components that can be modeled as FSMs. Each of these components is tested using well-proven techniques, such as the UIOv-method [13] or the Wp-method [4]. However, when these components are put together by adding glue code, no monolithic FSM is constructed in order to derive test cases for the composite system, which would lead to long test cases, large test suites, and a repetition of tests already performed on component level. Instead, the composite system is only tested for *composition faults*, i.e., faulty glue code, a new type of fault that extends and complements the classical fault model. We note that compositional testing is different from interoperability testing [1], where the objective is to check whether two implementations I_1 and I_2 conforming to the same specification S interoperate correctly. Here, two implementations may interoperate correctly, in some cases, and incorrectly, in other cases, because of faulty glue. Compositional testing is directed towards detecting faulty glue, assuming that I_1 and I_2 conform to S_1 and S_2, respectively.

In this paper, we will develop these ideas up to a certain point, and illustrate them through examples. We focus on a specific type of composition, called *concurrent composition*. However, other types of composition may be considered as well. The results so far may stimulate a new research stream that finally leads to a theory of compositional testing. Section 2 defines the concurrent composition of asynchronously communicating FSMs, and states necessary conditions for composability. The *compositional test method* (*C-method*) is defined in Section 3. An application is shown in Section 4. We draw conclusions and indicate future research topics in Section 5.

2. Concurrent composition

In this section, we define the concurrent composition of two FSMs. Further types of composition are perceivable, for instance, in the context of micro protocols [5]. On specification level, composition can be expressed by defining a composition operator. On implementation level, this operator is usually realized by a piece of code that we call *glue code*.

Concurrent composition may be applied to put local and/or remote components together. From the conceptual viewpoint, this should not make any difference. For instance, we may compose protocol entities $PE_{1,1}$ and $PE_{1,2}$ as well as $PE_{1,2}$ and $PE_{2,2}$ concurrently, as shown in Figure 1a) and b), respectively. For the local composition, the glue code may consist of internal data structures and operations to add signals to the input queue of the other protocol entity (Figure 1c). For the remote composition, the glue code may comprise an entire logical communication medium, which may in turn be a composite system (Figure 1d).

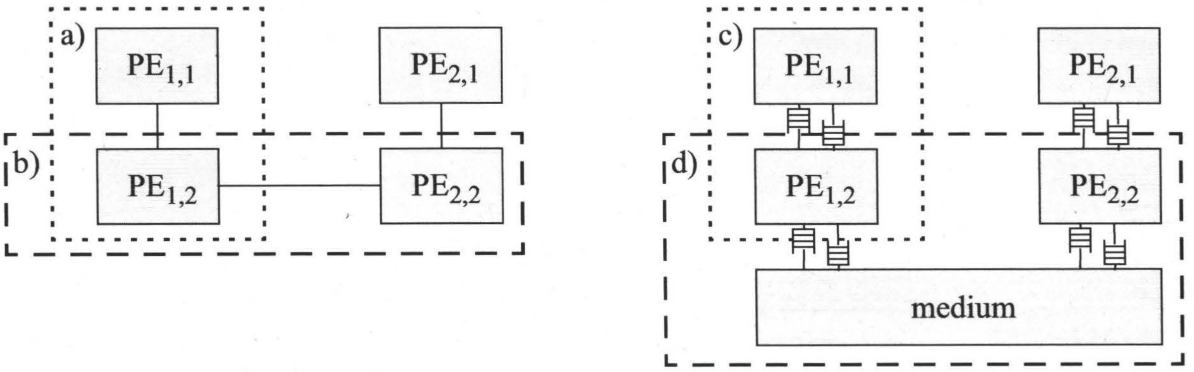

Figure 1: Concurrent composition of protocol entities

In this paper, we use the standard definition of FSM, and a derived notion:

Definition 1: A *finite state machine (FSM) M* is a tuple (S,I,O,s_0,λ_e) with:

- S is a finite set of states.
- I is a finite input alphabet.
- O is a finite output alphabet.
- $s_0 \in S$ is the initial state.
- $\lambda_e \subseteq S \times I \times O \times S$ defines the *transitions* of M.

A finite state machine is completely specified, if for each state and each input, a transition is defined. There exist several ways to extend a given FSM to a completely specified machine, e.g., by adding implicit transitions (cf. SDL [7]). Thus, FSMs are completely specified by default. We adopt this approach in the following definition:

Definition 2: A *completely specified finite state machine (csFSM)* $N = (S,I,O_e,s_0,\lambda)$ is derived from an FSM $M = (S,I,O,s_0,\lambda_e)$ as follows:

- S, I, s_0 as in M.
- $O_e = O \cup \{e\}$, where $e \notin O$ is called *error output*.
- $\lambda = \lambda_e \cup \lambda_i$, is the transition relation of N. Tuples of λ are called *transitions* of N.
- λ_e defines the *explicit transitions* of N.
- $\lambda_i = \{ (s,i,e,s) \mid s \in S \wedge i \in I \wedge \neg\exists o \in O, s' \in S: (s,i,o,s') \in \lambda_e \}$ defines the *implicit transitions* of N.

The standard definition of FSMs does not distinguish between explicit and implicit transitions. We consider explicit transitions as regular behavior. Implicit transitions are undesired behavior, but included to enhance testability of the implementation. Thus, csFSMs are always completely specified, i.e., for each state and each input, a transition is defined. The concept of implicit transition is similar to that found in SDL [7], with the difference that an error output is produced. Below, we omit the error output e and the transition relation λ_i for brevity.

To define the concurrent composition of csFSMs, we assume that they communicate by asynchronous reliable signal exchange, where sending and receiving of signals is modeled as output and input of the communicating csFSMs, respectively. Therefore, an input queue collecting signals that are delivered, but not yet consumed, is associated with each csFSM. Furthermore, each signal carries identifications of the sending and receiving machine, which may be evaluated as needed. The identifications are determined dynamically from the sending machine, the connection structure of the communicating csFSMs consisting of typed channels, and explicit addressing, if necessary. Again, this is similar to SDL.

Definition 3: Let $N_1 = (S_1,I_1,O_1,s_{0,1},\lambda_1)$ and $N_2 = (S_2,I_2,O_2,s_{0,2},\lambda_2)$ be csFSMs. Let $OI_{1,2} \subseteq O_1 \cap I_2$ ($OI_{2,1} \subseteq O_2 \cap I_1$) be the set of signals exchanged between N_1 and N_2 (N_2 and N_1), called *internal signals*. The *concurrent composition* of N_1 and N_2, denoted $N_1 \parallel N_2$, is defined by the derived state machine $Q = (S,I,O,s_0,\lambda)$:

- $S = S_1 \times I_1^* \times S_2 \times I_2^*$ is the set of states.
- $I = I_1 \cup I_2$ is the (finite) input alphabet.

- $O = O_1 \cup O_2$ is the (finite) output alphabet.
- $s_0 = (s_{0,1},<>,s_{0,2},<>)$ is the initial state, consisting of the initial states of N_1 and N_2 and the initial states of *input queues* associated with N_1 and N_2, respectively.
- $\lambda \subseteq S \times I \times O \times S$ is the transition relation of Q. Tuples of λ are called *transitions* of Q. λ is derived from λ_1 and λ_2 as follows:

  $(s,i,o,s') \in \lambda$ with $s = (s_1,q_1,s_2,q_2)$ and $s' = (s_1',q_1',s_2',q_2')$ iff

  $(\exists(s_1,i,o,s_1') \in \lambda_1\text{: } (q_1 = <i>^\frown q_{1'} \wedge q_2' = \textit{if } o \in OI_{12} \textit{ then } q_2^\frown<o> \textit{ else } q_2 \wedge s_2 = s_2')) \vee$

  $(\exists(s_2,i,o,s_2') \in \lambda_2\text{: } (q_2 = <i>^\frown q_{2'} \wedge q_1' = \textit{if } o \in OI_{21} \textit{ then } q_1^\frown<o> \textit{ else } q_1 \wedge s_1 = s_1'))$

This definition includes the concurrent composition of two independent csFSMs, i.e., two csFSMs that do not exchange signals. In this case, $OI_{1,2} = OI_{2,1} = \{\}$.

A csFSM can be represented as a labeled directed graph, where states correspond to nodes, and transitions correspond to edges labeled with input and output.

Definition 4: A *labeled directed graph* G is a tuple (V,L,E), consisting of a set of nodes V, a set of labels L, and a relation $E \subseteq V \times V \times L$, defining the directed edges of the graph. A *path* is a non-empty sequence of consecutive edges. A *tour* is a path that starts and ends at the same node. A directed graph G is *strongly connected*, if for each pair of nodes (v,v'), where $v \neq v'$, there is a path from v to v'.

Example 1: Figure 2 shows the concurrent composition of deterministic, strongly connected csFSMs N_1 and N_2. Note that the error output as well as the implicit transitions are not shown in the figure. The machines interact via *ch*, which is typed by $OI_{1,2}$ and $OI_{2,1}$, and are connected to the environment by typed channels ch_1 and ch_2. The resulting behavior after composition (see Figure 3) can be represented by the state machine $Q = N_1 \parallel N_2$, where states are represented as tuples (s_1,q_1,s_2,q_2) denoting the states of N_1 and N_2, and of their input queues.

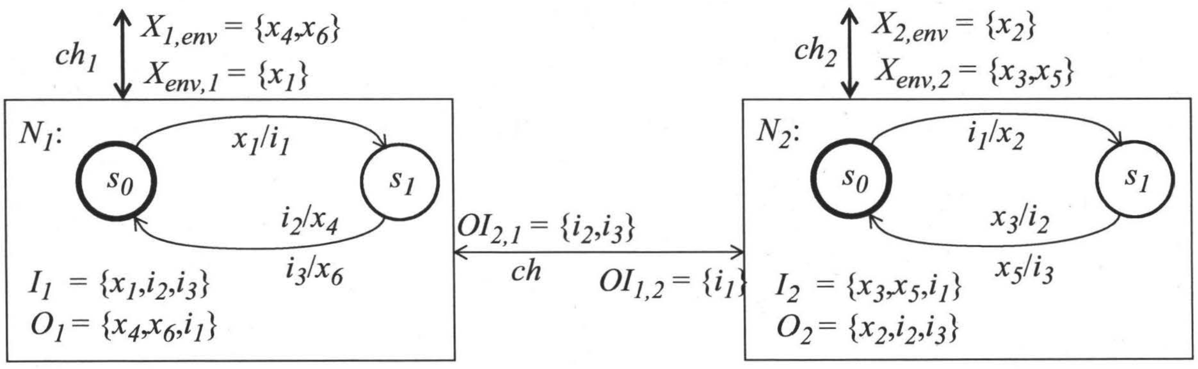

Figure 2: Concurrent composition: component machines N_1 and N_2 (Example 1)

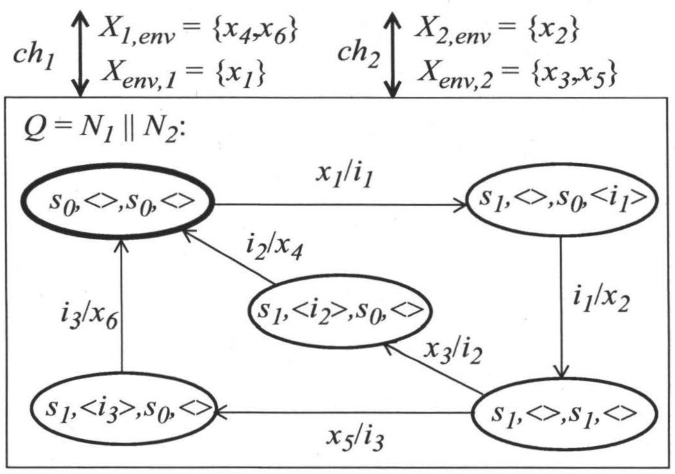
ch_1
$X_{1,env} = \{x_4, x_6\}$
$X_{env,1} = \{x_1\}$
ch_2
$X_{2,env} = \{x_2\}$
$X_{env,2} = \{x_3, x_5\}$
$Q = N_1 \| N_2$:
$s_0, \langle\rangle, s_0, \langle\rangle$
x_1/i_1
$s_1, \langle\rangle, s_0, \langle i_1\rangle$
i_2/x_4
i_1/x_2
i_3/x_6
$s_1, \langle i_2\rangle, s_0, \langle\rangle$
x_3/i_2
$s_1, \langle i_3\rangle, s_0, \langle\rangle$
$s_1, \langle\rangle, s_1, \langle\rangle$
x_5/i_3

Figure 3: Concurrent composition: derived machine *Q* (Example 1)

While it is syntactically possible to compose all kinds of csFSMs, this is not always meaningful. Which csFSMs to compose, first of all depends on the intended global behavior, which is problem specific. However, some *general composition criteria* can be stated:

CC_1. Internal signals of either machine are eventually consumed by the other machine in an *explicit* transition, i.e., the composed system is *free of internal unspecified receptions.* This excludes transitions that have been added to obtain a fully specified state machine, i.e., implicit transitions yielding an error output (see Definition 2).

CC_2. The composed system is free of *internal deadlocks*. Since it is assumed that external signals can be produced in any order, this again restricts the internal interaction only.

3. Compositional testing of concurrently composed csFSMs

In this section, we will show how to derive test suites for testing the implementation of concurrently composed csFSMs. We assume that certain assumptions concerning the component csFSMs (e.g., strongly connected, deterministic) and their implementations (e.g., concerning the number of states) are satisfied, and that the implementation of each csFSM can be tested using a test method that detects all output and transfer faults.

A direct approach to test the composite system would be to determine its global state machine, and then to apply one of the existing test methods to derive test cases for this machine. This, however, has the following drawbacks:

- The state set of the global state machine may be large. Firstly, this can consume considerable computational resources to determine the machine. Secondly, it can lead to a large test suite containing long test cases, implying a testing effort that quickly becomes unmanageable.
- The global state machine may be non-deterministic, due to the concurrency of the composite system, which reduces the applicability of existing test methods.

- All tests already executed on component level are in fact repeated. This is a severe drawback in general, and especially if components are to be reused in different protocol configurations.

To avoid these disadvantages, a test method satisfying the following properties is sought:

- It is not necessary to compute the global state machine.
- Only tests checking the correctness of the glue code of the csFSMs are derived.
- Tests already performed on component level are not repeated.

These properties can only be satisfied if the implementations of the design components, which have been tested on component level, remain unchanged. This means that only glue code to realize the specific type of composition is added. Only if this constraint is satisfied, tests that have been applied on component level (e.g., to detect transfer and output errors) need not be repeated after composition. All that remains to be checked is the correct implementation of the composition operator.

In the following, we introduce a method for compositional testing - henceforth called *compositional test method* (*C-method*) - that satisfies the above properties. We start by defining the fault model, then introduce concepts, notations, and an initial tour coverage graph, and finally give a procedural definition of the C-method.

3.1. Fault model

The common way to check that a conformance relation that is defined on an infinite set of input sequences holds between two finite state machines is to reduce the set of possible implementations to a finite number by assuming a fault model [10]. The classical fault model of protocol testing assumes that the implementation *I* can be treated as a *mutant* of the specification *S*, where a mutant may be obtained by altering outputs of transitions (*output faults*), by altering tail states of transitions (*transfer faults*), by adding states up to a given number as well as extra transitions to and from these states. This general fault model is sometimes reduced to output and transfer faults by assuming that the number of implementation states is less or equal to the number of specification states, and to deterministic implementations.

Implementations are tested by applying input sequences and observing the output sequences. An implementation fault is detected, if an observed output sequence differs from the expected output sequence. Whether this fault is an output fault or a transfer fault, or due to an extra state or an extra transition, depends on the fault model, on the diagnosis capability of the test method, and on the knowledge about the implementation at the time of test execution.

The classical fault model is usually applied to single components that are specified by a finite state machine, e.g., a single protocol entity. It may also be applied to a composite system, e.g., protocol entities and an underlying medium, if a finite state machine of that system can be constructed. This, however, causes the aforementioned problems (large state spaces, nondeterminism, repetition of tests). In order to avoid these problems, we propose to take the structural aspect of the composition into account, and to distinguish the following fault categories:

- *component fault*: the implementation of a *component* does not satisfy its specification
- *composition fault*: the *glue code* does not satisfy its specification in the given context

The problem of compositional testing can then be stated as follows:

Let N_1 and N_2 be the specifications of two components, and I_1 and I_2 be their implementations, where I_1 and I_2 satisfy their specifications N_1 and N_2, respectively. Then, derive a minimal test suite that is sufficient to check whether the system I consisting of I_1, I_2, and glue code satisfies the specification $N_1 \parallel N_2$.

As before, implementations are tested by applying input sequences, and comparing the observed and the expected output sequences. Again, it depends on the fault model, the diagnosis capability of the test method, and the knowledge about the implementation at the time of test execution how a detected fault may be classified. For instance, if the components have already been tested successfully, and their implementations are reused in the composite system, then detected faults can be classified as composition faults.

To derive a minimal test suite that is sufficient to check the composed system, a model of the glue code is needed. In general, the glue code could be a component or a composite system itself, for instance, a logical communication medium, which may have further attached components. As testing would be unfeasible in this general setting, we make the following assumption:

i) Whenever I_1 and I_2 are both in their initial states, the glue code is in a determined state w.r.t. I_1 and I_2.

ii) The behavior of the glue code is deterministic w.r.t. I_1 and I_2.

iii) If the glue code interacts with other components, this has no effect on its behavior towards I_1 and I_2.

iv) The glue code is not creating messages for N_1 or N_2.

The first assumption limits the maximum length of test suites to the set of all initial tours, i.e., paths that start and end in the initial state. All assumptions together ensure that a finite number of test cases is sufficient.

3.2. Concepts and notations

The following definitions recall and introduce some concepts and notations for testing:

Definition 5: A *test case tc* is a non-empty sequence of inputs $i_1.i_2.....i_n$. A *test suite ts* is a non-empty set of test cases $\{tc_1,tc_2,...,tc_m\}$. An *augmented test case atc* is defined as a non-empty sequence of transitions (also called *test elements*) $i_1/o_1.i_2/o_2.....i_n/o_n$. An *augmented test suite ats* is a non-empty set of augmented test cases $\{atc_1,atc_2,...,atc_m\}$.

Definition 6: Let atc_1 and atc_2 be augmented test cases (sequences of transitions) of deterministic csFSMs N_1 and N_2 that communicate via a common channel *ch* with sets $OI_{1,2}$ and $OI_{2,1}$ of internal signals. The *concurrent composition of* atc_1 *and* atc_2, denoted $atc_1 \parallel atc_2$, is obtained by combining all *test elements* into one augmented test case $atc_{1,2}$ such that the following constraints are satisfied:

- the order of test elements of atc_1 and atc_2 is preserved;

- for atc_1 (atc_2), test elements triggered by external signals may be added to $atc_{1,2}$ without further constraints;
- for atc_1 (atc_2), test elements by internal signals may only be added to $atc_{1,2}$ if this signal has already been output in a previous test element, and is ready for consumption;
- the order of outputs is preserved.

The rationale here is that internal signals are entirely controlled by the concurrently composed csFSMs, while external signals are under the control of the tester and can therefore be sent at any time. From the definition, it follows that || commutes, i.e., $atc_{1,2} = atc_{2,1}$.

Example 2: For the csFSMs N_1 and N_2 of Example 1, the following augmented test cases can be derived and composed:

- $atc_1 = x_1/i_1.i_2/x_4$
- $atc_2 = i_1/x_2.x_3/i_2$
- $atc_1 \,||\, atc_2 = x_1/i_1.i_1/x_2.x_3/i_2.i_2/x_4$

Definition 7: The concurrent composition of two augmented test cases is called *complete*, iff all their test elements are included, and the input queues of the corresponding csFSMs will be empty after their execution. Otherwise, it is called *incomplete*.

Example 3: The concurrent composition of atc_1 and atc_2 in Example 2 is complete. However, the concurrent composition of atc_1 and $atc_{2'} = i_1/x_2$ results in $x_1/i_1.i_1/x_2$, which is incomplete.

3.3. Initial tour coverage tree

Selected augmented test cases of components form the basis for deriving a test suite for validating the correct implementation of their composition. These test cases are derived from a so-called initial tour coverage tree, reduced to the set of relevant test cases, and composed with matching test cases of the other component.

Definition 8: Let $N = (S,I,O_e,s_0,\lambda)$ be a csFSM with the underlying graph G, where G is strongly connected. An *initial tour of G* is a tour of the underlying graph that starts and ends in the node corresponding to the initial state s_0 of N. It is called *minimal*, if no edge is contained more than once. A *complete set of minimal tours* is a set of minimal tours such that every edge is covered at least once.

Definition 9: Given a csFSM $N = (S,I,O_e,s_0,\lambda)$, where the underlying graph is strongly connected, an *initial tour coverage tree T* is a tree that covers all minimal tours.

The rationale behind this choice is that (i) transition coverage can be achieved this way[1], and that (ii) both automata should be synchronized at least in their initial states, a criterion for composability. To construct an initial tour coverage tree, we use a tree that, for a given state, captures all cycle free paths to the initial state, called *homing tree*:

1. Initial tour coverage is a reduced form of path coverage.

Definition 10: Given a csFSM $N = (S,I,O_e,s_0,\lambda)$ and a state $s \in S$, where the underlying graph is strongly connected, a *homing tree* $H(s)$ is a minimal tree that covers all cycle-free paths of N leading from s to the initial state s_0.

We give algorithms for the construction of homing trees and initial tour coverage trees in Tables 1 and 2, respectively. Both algorithms are illustrated.

Step 1: Start the construction of $H(s)$ with its root node n_r, labeled with s.

Step 2: Assume that $H(s)$ has been constructed up to level k, $k \geq 1$. Then level $k+1$ is built by examining the nodes of level k:

- Step 2.1: A node n of level k is terminated, if its label is identical to the label of a node on level j, where $1 \leq j < k$, or if it is identical to s_0.
- Step 2.2: Otherwise, let s denote the label of node n. Then, for all transitions (s,x,y,s'), attach a branch and successor node to the current node, labeled x/y and s', respectively.

Step 3: Prune the resulting tree by successfully removing all leaf nodes that have a label $s \neq s_0$, and the corresponding edges.

Table 1: Construction of a homing tree $H(s)$

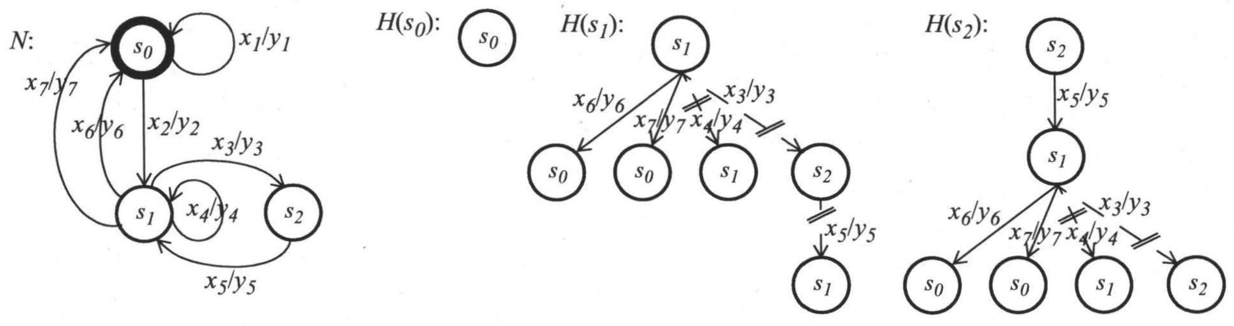

Figure 4: Homing trees (example)

Step 1: For each state s of N, construct a homing tree $H(s)$.

Step 2: Start the construction of T with the root node n_r, labeled with the initial state s_0 of N. This is level 1 of T.

Step 3: Assume that T has been constructed up to level k, $k \geq 1$. Then level $k+1$ is built by examining the nodes of level k:

- Step 3.1: A node n of level k is terminated, if its label is identical to the label of a node on level j, where $1 \leq j < k$.
- Step 3.2: Otherwise, let s denote the label of n. Then, for each transition (s,x,y,s'), attach a branch and successor node to the current node, labeled x/y and s', respectively.

Step 4: To each leaf node n, attach the homing tree $H(s)$ by merging the root node of $H(s)$ with n, where s denotes the label of n.

Table 2: Construction of an initial tour coverage tree T

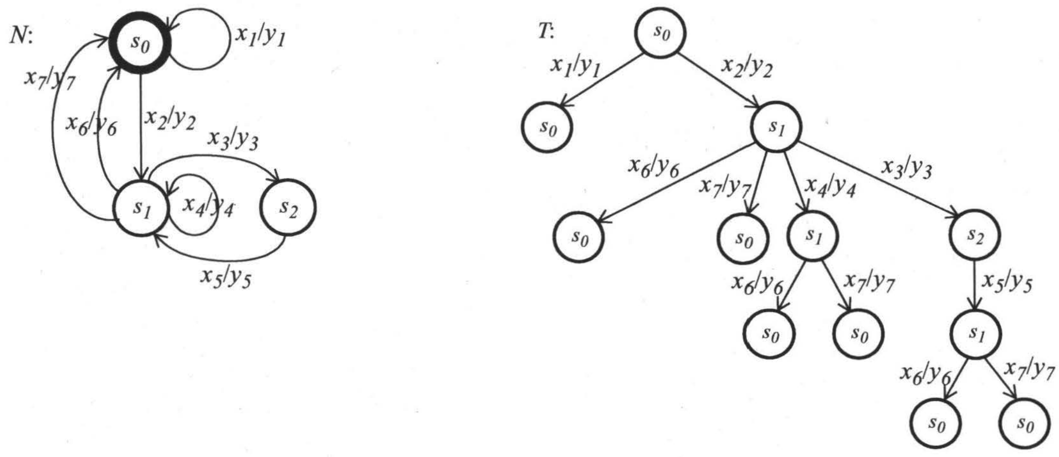

Figure 5: Initial tour coverage tree (example)

3.4. The C-method

In Section 2, we have stated general composition criteria CC_1 and CC_2 that should be satisfied for a meaningful composition on design level. First, the composed system should be free of internal unspecified receptions, which means that receptions occurring during „normal operation" have to be consumed by regular transitions. This excludes transitions that have been added for mere technical reasons to obtain fully specified state machines (see Definition 2). Also, the composed system should be free of internal deadlocks.

C-method

Step 1: Test the implementations I_1 and I_2 of components N_1 and N_2.

Step 1.1: Select a suitable test method (e.g., DS [8], UIOv [13], Wp [9]).

Step 1.2: Derive the test suites for N_1 and N_2.

Step 1.3: Execute the tests. If all tests are successful, continue with Step 2. If not, correct the faults and repeat Step 1.

Step 2: Test the implementation of the concurrent composition of N_1 and N_2.

Step 2.1: Remove all transitions of N_1 and N_2 that yield an error output. These transitions have already been tested during component testing, and need not be tested again.

Step 2.2: Build the initial tour coverage trees for N_1 and N_2, and determine all maximal paths, i.e., all paths that start at the root node and end at a leaf node, constituting augmented test suites ats_1 and ats_2.

Step 2.3: From the augmented test suites ats_1 (ats_2), remove all internally triggered test cases, i.e., those test cases that are triggered by N_2 (N_1).

Step 2.4: From the augmented test cases, remove all local tours, i.e., (sub)sequences of test case elements that (1) start and end in the same state, and (2) contain only external inputs and outputs. They have already been checked during component testing, and need not be repeated.

Step 2.5: Remove the maximum suffix that does not contain an interaction with the other component. These test elements have been checked already.

Step 2.6: For each test case $atc_{1,j}$ of the augmented test suite ats_1 after Step 2.5, find an augmented test case $atc_{2,j}$ of N_2 that starts and ends in the initial state such that $atc_{1,j} \| atc_{2,j}$ is complete, and determine $atc_{1,2,j} = atc_{1,j} \| atc_{2,j}$, yielding the concurrent augmented test suite $ats_{1,2}$. Analogously for each test case $atc_{2,j}$ of ats_2.

Step 2.7: Based on ats_1, ats_2, and $ats_{1,2}$, check whether N_1 and N_2 meet the composition criteria CC1 and CC2, i.e., whether for each test case of ats_1 (ats_2), there is a matching test case of N_2 (N_1). Yes: continue with Step 2.8; no: stop.

Step 2.8: For each test case in $ats_{1,2}$: merge adjacent test case elements in cases where (1) the internal output of the first matches the internal input of the second, and (2) the output is the only signal in the queue after being sent. Replace internal inputs and outputs by "-", and remove test case elements "-/-".

Step 2.9: Execute the test.

Table 3: The C-method

To check whether two csFSMs N_1 and N_2 meet these criteria, we assume that they are always capable to resynchronize in their initial states. In other words, if N_1 is in its initial state and stays there, N_2 should be able to reach its initial state without further interaction with N_1, and vice versa. If this assumption is satisfied, it suffices to consider the explicit initial tours of both automata, i.e., the explicit transition sequences starting and ending in the initial states, and to check whether for each explicit initial tour, there is a matching explicit initial tour of the other automaton such that their concurrent composition is complete. This design criterion can also be stated in terms of concurrent composition of augmented test suites, and thus be checked as a by-product of test case derivation.

In Table 3, the C-method is defined in a procedural style. We point out that in the course of applying the test procedure, it is checked whether N_1 and N_2 satisfy the composition criteria. This is a constraint imposed on design level, which should be checked before implementing the design and testing the implementation. Thus, all steps except Steps 1.2, 1.3, 2.8, and 2.9 should already be executed in the design phase.

As expected, the augmented test suites ats_1 and ats_2 are reduced to empty test suites in case N_1 and N_2 do not interact, i.e., in case of their independent concurrent composition, which, among other things, satisfies the criterion for concurrent composability. The reason is that all necessary testing has already been done on component level. Of course, one can argue that in the implementation, interaction of the two components may occur, and has to be excluded. This, however, is not covered by this type of tests. When protocol components are reused, it is sufficient to test them once, which means in a certain sense that testing is reused, too. In these cases, compositional testing can start with Step 2.

4. Application of the C-method to the InRes protocol

To illustrate the C-method, we apply it to the Initiator Responder (InRes) protocol [6]. The InRes protocol is a connection-oriented communication protocol for the reliable exchange of message over an order-preserving, connection-less medium. It provides an asymmetrical service: the initiator requests connections and sends data, the responder accepts, refuses, and clears connections, and receives data. In this example, the InRes protocol entities *I* and *R* are the components that are composed concurrently, yielding a composite system $I \parallel R$. In the implementation of this system, the glue code is represented by the underlying medium. To be able to use this medium for the implementation of the $I \parallel R$, we assume that it does not lose messages.

Figure 6 shows the specifications *I* and *R* of the the InRes protocol entities and their concurrent composition. Both automata contain further transitions that can be derived by applying Definition 2, and thus are fully-specified. To avoid cluttering, we have omitted these transitions in the figure. The underyling graphs are deterministic, and strongly connected. We assume that Step 1 of the C-method that tests the implementations of *I* and *R* separately has already been executed successfully. Below, we go through Step 2:

- Step 2.1: Removal of transitions yielding an error output

  These transitions have been omitted in the figure, therefore, starting point for Step 2.2 are the finite state automata shown in Figure 6.

- Step 2.2: Build initial tour coverage trees, and determine ats_I and ats_R
  The initial tour coverage trees for *I* and *R* are shown in Figure 7. The test suites are:

  ats_I = $\{atc_{I,1}, atc_{I,2}, atc_{I,3}, atc_{I,4}\}$, with
  $atc_{I,1}$ = ICONreq/CR . DR/IDISind
  $atc_{I,2}$ = ICONreq/CR . CC/ICONcnf . DR/IDISind
  $atc_{I,3}$ = ICONreq/CR . CC/ICONcnf . IDATreq/DT . DR/IDISind
  $atc_{I,4}$ = ICONreq/CR . CC/ICONcnf . IDATreq/DT . AK/- . DR/IDISind

  ats_R = $\{atc_{R,1}, atc_{R,2}, atc_{R,3}, atc_{R,4}\}$
  $atc_{R,1}$ = DT/-
  $atc_{R,2}$ = CR/ICONind . IDISreq/DR
  $atc_{R,3}$ = CR/ICONind . ICONrsp/CC . IDISreq/DR
  $atc_{R,4}$ = CR/ICONind . ICONrsp/CC . DT/IDATind . -/AK . IDISreq/DR

- Step 2.3: Remove test cases triggered by internal inputs
  All test cases of *R* are triggered by inputs of the Initiator and therefore removed:
  ats_I = $\{atc_{I,1}, atc_{I,2}, atc_{I,3}, atc_{I,4}\}$
  ats_R = $\{\}$

- Step 2.4: Remove external local tours
  Not applicable in the InRes example.

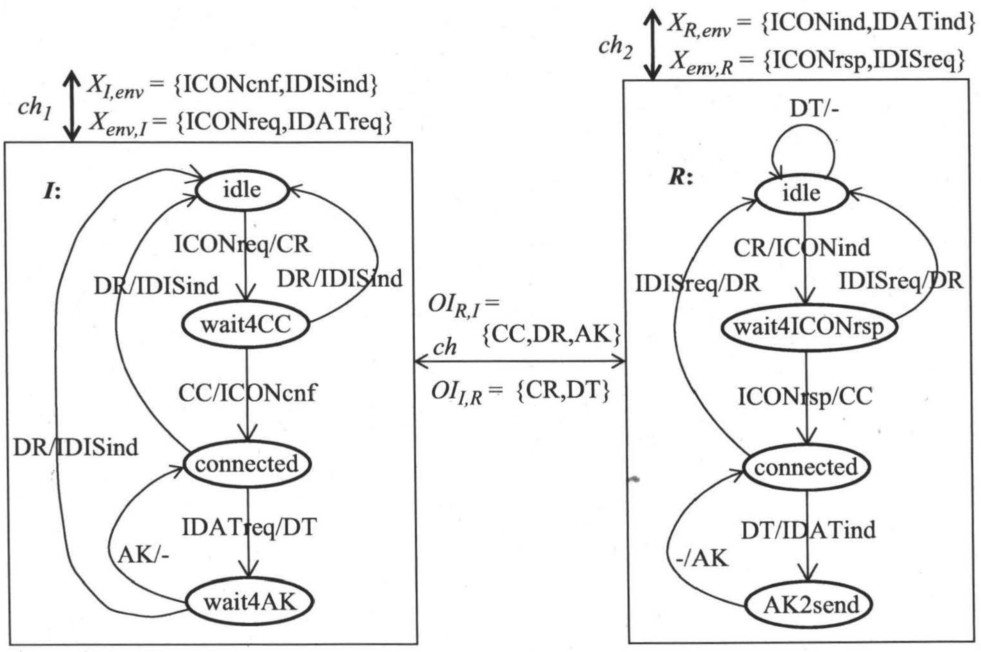

Figure 6: InRes protocol entities *I* and *R*

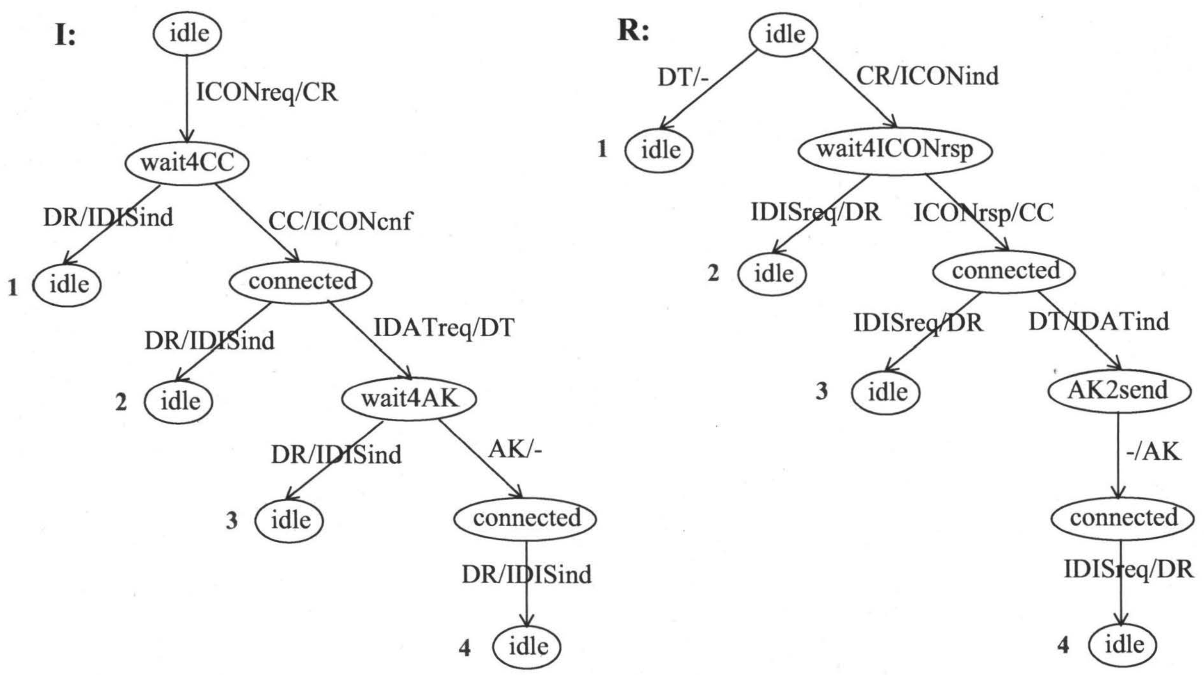

Figure 7: Initial tour coverage graphs of *I* and *R*

- Step 2.5: Remove suffix containing external interaction only.
  Not applicable in the InRes example.

- Step 2.6: For each augmented test case in ats_I (ats_R), find an augmented test case from *R* (*I*) such that their concurrent composition is complete, and determine the concurrent augmented test suite $ats_{I,2}$.

  $ats_{I,2}$ = { $atc_{I,1} \| atc_{R,a}$, $atc_{I,2} \| atc_{R,b}$, $atc_{I,3} \| atc_{R,c}$, $atc_{I,4} \| atc_{R,d}$ }, with:

  $atc_{I,1}$ = ICONreq/CR . DR/IDISind

  $atc_{R,a}$ = CR/ICONind . IDISreq/DR

  $atc_{I,1} \| atc_{R,a}$ = { ICONreq/CR . CR/ICONind . IDISreq/DR . DR/IDISind }

  $atc_{I,2}$ = ICONreq/CR . CC/ICONcnf . DR/IDISind

  $atc_{R,b}$ = CR/ICONind . ICONrsp/CC . IDISreq/DR

  $atc_{I,2} \| atc_{R,b}$ = { ICONreq/CR . CR/ICONind . ICONrsp/CC . CC/ICONcnf .
  IDISreq/DR . DR/IDISind,
  ICONreq/CR . CR/ICONind . ICONrsp/CC . IDISreq/DR .
  CC/ICONcnf . DR/IDISind }

  $atc_{I,3}$ = ICONreq/CR . CC/ICONcnf . IDATreq/DT . DR/IDISind

  $atc_{R,c}$ = CR/ICONind . ICONrsp/CC . IDISreq/DR . DT/-

  $atc_{I,3} \| atc_{R,c}$ = { ICONreq/CR . CR/ICONind . ICONrsp/CC . CC/ICONcnf .

IDATreq/DT . IDISreq/DR . DR/IDISind . DT/-,
ICONreq/CR . CR/ICONind . ICONrsp/CC . CC/ICONcnf .
IDATreq/DT . IDISreq/DR . DT/- . DR/IDISind,
ICONreq/CR . CR/ICONind . ICONrsp/CC . CC/ICONcnf .
IDISreq/DR . IDATreq/DT . DT/- . DR/IDISind,
ICONreq/CR . CR/ICONind . ICONrsp/CC . IDISreq/DR .
CC/ICONcnf . IDATreq/DT . DT/- . DR/IDISind }

$atc_{I,4}$ = ICONreq/CR . CC/ICONcnf . IDATreq/DT . AK/- . DR/IDISind

$atc_{R,d}$ = CR/ICONind . ICONrsp/CC . DT/IDATind . -/AK . IDISreq/DR

$atc_{I,4} \| atc_{R,d}$ = { ICONreq/CR . CR/ICONind . ICONrsp/CC . CC/ICONcnf .
IDATreq/DT . DT/IDATind . -/AK . AK/- . IDISreq/DR . DR/IDISind,
ICONreq/CR . CR/ICONind . ICONrsp/CC . CC/ICONcnf .
IDATreq/DT . DT/IDATind . -/AK . IDISreq/DR . AK/- . DR/IDISind}

- Step 2.7: Check the composition criteria CC1 and CC2
  For each test case of ats_I, there is a test case of R such that their concurrent composition is complete. This trivially holds for ats_R, which is empty.

- Step 2.8: Merge test case elements, and replace internal inputs and outputs by „-“
  $ats_{1,2}$ = { $atc_{I,1} \| atc_{R,a}$, $atc_{I,2} \| atc_{R,b}$, $atc_{I,3} \| atc_{R,c}$, $atc_{I,4} \| atc_{R,d}$ }, with:
  $atc_{I,1} \| atc_{R,a}$ = { ICONreq/ICONind . IDISreq/IDISind }
  $atc_{I,2} \| atc_{R,b}$ = { ICONreq/ICONind . ICONrsp/ICONcnf . IDISreq/IDISind }
  $atc_{I,3} \| atc_{R,c}$ = { ICONreq/ICONind . ICONrsp/ICONcnf . [IDATreq/- ||| IDISreq/-] . -/IDISind }
  $atc_{I,4} \| atc_{R,d}$ = { ICONreq/ICONind . ICONrsp/ICONcnf . IDATreq/IDATind . IDISreq/IDISind }

Note that test case $atc_{I,3} \| atc_{R,c}$ requires that test input IDATreq and IDISreq are to be applied concurrently to stimulate this behavior. This is expressed by the notation [tce_1 ||| tce_2]. The resulting test suite $ats_{1,2}$ consists of 4 test cases, with 14 test case elements. In addition, component tests are to be performed.

5. Conclusions and future work

In this paper, the *compositional method* (*C-method*) for testing communication protocols has been introduced. The C-method first tests each protocol component separately for component faults (output and/or transfer faults), using one of the traditional test methods, and then checks their composition for composition faults.

To apply the C-method, it is not necessary to compute the global state machine. Instead, composition tests are derived from local initial tour coverage trees. Only tests checking the glue code are derived. We have introduced and justified a fault model for the glue code that leads to manageable composition test suites.

The work on compositional testing has been triggered by our results on micro protocols [5], a concept to structure communication systems and to foster reuse of protocol designs. Micro protocols are protocols with a single (distributed) functionality and the required collaboration among protocol entities. To develop customized communication systems, micro protocol designs are selected from a library, composed to yield a complete design, and implemented. We use the ITU language SDL to formally specify micro protocol designs, and to compose them. We believe that the C-method will contribute to the testing of customized communication systems that are composed of micro protocols.

The results presented in this paper leave room for further work. The following improvements and enhancements are perceivable:

- The justification of the C-method and its benefits should be treated more rigorously, developing a test theory that is rich enough to provide a formal proof that the derived test suite is both necessary and sufficient to detect all composition faults.
- So far, only the concurrent composition of two FSMs has been considered. It would be useful to extend the C-method to compositions of more than two FSMs, and also to the composition of composites that have already been tested successfully.
- Other types of compositions, for instance, concurrent composition with shared variables, or composition through inheritance, are perceivable. Again, this requires extensions to the C-method.

In conclusion, the presented results may stimulate a new research stream that eventually may lead to a theory of compositional testing of asynchronous systems.

References

[1] R. Castanet, O. Kone: *Deriving Coordinated Testers for Interoperability*, Protocol Test Systems, Volume VI C-19, Pau, France, 1994

[2] T. S. Chow: *Testing Software Design Modeled by Finite-State Machines*, IEEE Transactions on Software Engineering, Vol. SE-4, No. 3, 1978, pp. 178-187

[3] A. En-Nouaary, R. Dssouli, F. Khendek: *Timed Wp: Testing Real-Time Systems*, IEEE Transactions on Software Engineering, 2002, pp. 1023-1038

[4] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, A. Ghedamsi: *Test Selection Based on Finite State Models*, IEEE Transactions on Software Engineering, Vol. 17, No. 6, June 1991, pp. 591-603

[5] R. Gotzhein, F. Khendek, P. Schaible: *Micro Protocol Design: The SNMP Case Study*, SDL and MSC Workshop (SAM'2002), Aberystwyth, UK, June 24-26, 2002

[6] D. Hogrefe: *OSI Formal Specification Case Study: The InRes Protocol and Service, revised*, Report No. IAM-91-012, Update May 1992, University of Berne, May 1992

[7] ITU-T Recommendation Z.100 (11/99) - *Specification and Description Language (SDL)*, International Telecommunication Union (ITU), 1999

[8] Z. Kohavi: *Switching and Finite Automata Theory*, McGraw Hill, USA, 1978

[9] G. Luo, G. v. Bochmann, A. Petrenko: *Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method*, IEEE Transactions on Software Engineering, SE-20, No. 2 (1994), pp. 149-162

[10] A. Petrenko, G. v. Bochmann, M. Yao: *On Fault Coverage of Tests for Finite State Specifications*, Computer Networks and ISDN Systems, Special Issue on Protocol Testing, Vol. 29, 1996, pp. 81-106

[11] B. Sarikaya, G. v. Bochmann: *Some Experience with Test Sequence Generation for Protocols*, Proceedings of the 2nd International Workshop on Protocol Specification, Testing, and Verification, North Holland, 1982, pp. 555-567

[12] B. Sarikaya, G. v. Bochmann: *Synchronization and Specification Issues in Protocol Testing*, IEEE Transactions on Communications, COM-32, No. 4, 1982, pp. 389-395

[13] S. T. Vuong, W. W. L. Chan, M. R. Ito: *The UIOv-Method for Protocol Test Sequence Generation*, Second International Workshop on Protocol Test Systems, Berlin, Germany, 1989