Automatic Generation of Test Purposes for Testing Distributed Systems

Olaf Henniger[1], Miao Lu[2], and Hasan Ural[2]

[1] Fraunhofer Institute Secure Telecooperation,
Rheinstr. 75, 64295 Darmstadt, Germany
henniger@sit.fraunhofer.de
[2] School of Information Technology and Engineering, University of Ottawa,
Ottawa, Ontario, K1N 6N5, Canada
{mlu,ural}@site.uottawa.ca

Abstract. In this paper, we present an algorithm for generating test purpose descriptions in form of MSC's from a given labeled event structure that represents the behavior of a system of asynchronously communicating extended finite state machines. The labeled event structure is a non-interleaving behavior model describing the behavior of a system in terms of the partial ordering of events.

1 Introduction

For testing whether the behavior of an implementation conforms to its designated behavior, test cases are to be generated from the specification describing the designated behavior. The behavior of a distributed system can be specified e.g. using a system of asynchronously communicating state machines. This model forms the basis e.g. of the standardized formal description technique SDL [1]. A system of communicating state machines implicitly describes all, possibly non-deterministic, sequences of inputs and outputs that constitute the designated behavior. Since the number and length of these sequences are infinite in general, it is impossible to test each and every possible behavior and we face the problem to select a set of meaningful test cases, i.e. a test suite, that allows to discover as many implementation errors as possible at an acceptable cost. This forms the main problem in generating conformance test suites.

Each test case in a test suite specifies the actions required to achieve a specific test purpose. The test purpose in each case is to check a particular requirement implied by the given specification [2]. A test purpose can be expressed e.g. by prose text or by a message sequence chart (MSC) describing the behavior to be checked. MSC's are a standardized description technique for the graphical representation of the temporal ordering of interactions between components of a distributed system [3].

The existing methods for test generation from formal specifications can be roughly classified into methods with explicit test purposes and methods with implicit test purposes. Methods with explicit test purposes require information about the test purposes as input in addition to the specification. These methods offer much flexibility to the test designer and ensure that only executable test cases are generated. However, they require considerable manual efforts to define appropriate test purposes and do not guarantee systematic test coverage. Methods with implicit test purposes provide test cases for test purposes that they tacitly assume. These methods generally guarantee a complete test coverage w.r.t. the implicit test purposes. However, most of them are applicable only to restricted classes of specifications, e.g. to specifications containing a single state machine, and they may result in very large test suites.

Since practically relevant system specifications may be voluminous and complicated, a manual generation and maintenance of test purposes and test cases is too time-consuming and error-prone. It is therefore highly desirable to have test generation methods with implicit test purposes or at least methods for the automatic generation of test purposes. Only few test generation tools, like Autolink [4] in the Telelogic Tau toolset and TestComposer [5] in the Telelogic ObjectGeode toolset are applicable to complex multi-process SDL specifications of a realistic size. These two tools are based on interleaving models for the behavior of the specified system. This entails that the same behavior may be represented by different paths of the reachability graph, which differ only in the order of execution of concurrent actions.

Our approach uses a non-interleaving model (labeled event structure) to alleviate the state-space explosion problem. In [6], an algorithm for transforming a system of asynchronously communicating state machines into a labeled event structure is given and a method with implicit test purposes for generating test cases in Concurrent TTCN from a labeled event structure is proposed. To combine the advantages of methods with implicit test purposes with those of methods with explicit test purposes, this paper aims at the automatic generation of test purposes from labeled event structures. From a labeled event structure, test purpose descriptions are generated in form of MSC's by interpreting the parallel paths of the labeled event structure as MSC's. These MSC's can serve as input for test generation tools with explicit test purposes, preferably if those support test generation for distributed testers, as proposed in [7, 8].

The rest of this paper is organized as follows. Section 2 introduces the prerequisites necessary for the proposed approach. Section 3 deals with the generation of test purposes from a labeled event structure. Throughout the paper, a simple sliding-window protocol serves as an example. Section 4 gives a summary and outlook.

2 Preliminaries

2.1 Communicating State Machines

A system of asynchronously communicating state machines is an obvious semantic model for specifications in SDL. Therefore, they form the starting point for our approach.

A *system of asynchronously communicating state machines* is composed of a set of state machines and a set of perfect (i.e. without loss or reordering of messages) FIFO queues that connect the state machines with each other and with their environment.

We consider each state machine as an extended finite state machine (EFSM) without enabling conditions for transitions. In general, an EFSM is a finite state machine extended by additional variables that may be used in enabling conditions for transitions, in calculations to be carried out during the execution of transitions, or for representing message parameters. An EFSM with enabling conditions can be transformed into an equivalent one without enabling conditions if the variables influencing the executability of transitions take on only a finite number of discrete values. An algorithm for this transformation is given in [9, 6]. This condition is not unduly restricting the class of specifications for which the algorithm for generating test purposes is applicable since it is a common practice for a test designer to determine the context by assigning values to control variables and to parameters of input messages.

We do not require that the EFSM's form a closed system, but allow open interfaces to the environment. To limit the complexity imposed by the environment, the following assumption is made. The environment is assumed to put a message into a queue if and only if the associated EFSM is ready to consume it. Hence, a transition with a trigger input (excited by a message from the environment) is assumed always to be enabled as soon as the EFSM reaches the start state of that transition. This assumption is common practice in test generation for conformance testing, which is, in contrast to robustness testing, confined to the behavior foreseen in the specification.

Let $m = \langle M, Q \rangle$ be a system of asynchronously communicating EFSM's composed of a set of EFSM's $M = \{m_1, \ldots, m_n\}$ and a set of message queues $Q = \{q_1, \ldots, q_r\}$. A *global state* of m is an $(n+r)$-tuple $g = (s_{m_1}, \ldots, s_{m_n}, c_{q_1}, \ldots, c_{q_r})$ consisting of the states $s_{m_1}, \ldots, s_{m_n}$ of the EFSM's $m_1, \ldots, m_n$ and the contents $c_{q_1}, \ldots, c_{q_r}$ of the queues $q_1, \ldots, q_r$.

Fig. 1 shows a system of asynchronously communicating EFSM's modeling a simple sliding-window protocol. The EFSM's *t*, *r*, and *m* model the transmitter and receiver protocol entities and the transmission medium, respectively. To facilitate de-

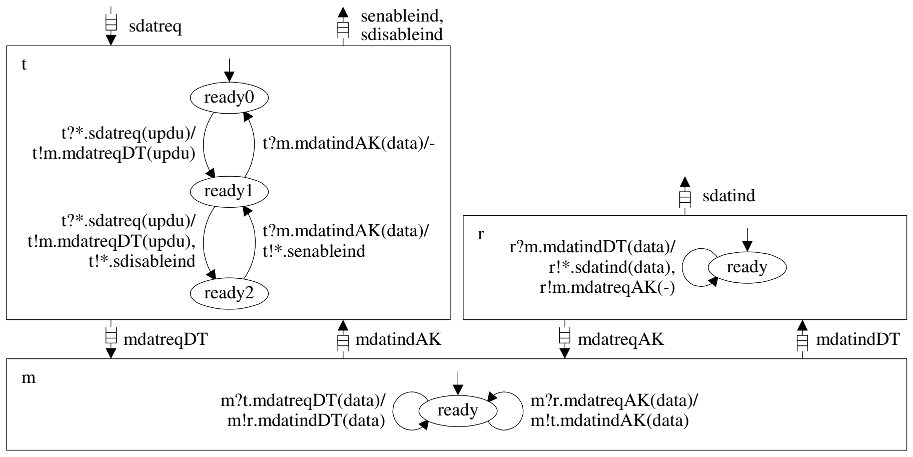

Fig. 1. Example of a system of asynchronously communicating EFSM's

nominating the location of actions, we denote input and output actions in the form *loc* ("?" | "!") *rem* "." *msg* ["(" *par* {"," *par*} ")"] where *loc* denotes the EFSM where the action is located, "?" indicates an input action (receiving *msg*), "!" indicates an output action (sending *msg*), *rem* is the name of the remote EFSM sending *msg* (in case of an input action) or receiving *msg* (in case of an output action), *msg* is a message, and *par* is a message parameter. "*" stands for the environment.

The example protocol provides the service to transmit data from a user on the transmitter side to a user on the receiver side while protecting the receiver against overload by attending to acknowledgements. If the number of messages for which the acknowledgement is outstanding (the window size) reaches its maximum (2 for simplicity), the protocol entity on the transmitter side indicates to the user that no more messages can be transmitted for the time being. When the protocol entity on the transmitter side receives an acknowledgement, then the number of messages for which the acknowledgement is outstanding is decremented and new messages can be transmitted again. The transmission medium is reliable and does not lose, corrupt, add, or reorder messages.

2.2 Labeled Event Structures

Definitions. For generating test purposes, we would like to have a model that explicitly describes the behavior of a distributed system in terms of the order of events. A labeled event structure fulfils this requirement. A system of asynchronously communicating EFSM's can be "unfolded" into a labeled event structure. In a labeled event structure concurrent events are not linearized as in a reachability tree, but lined up side by side without order relation. Event structures were introduced in [10] as being like acyclic Petri nets without backward branching and with the places removed.

A basic element of labeled event structures are actions. The same action can occur various times in a system run, each time forming a new, distinguishable event. The actions in the labeled event structures correspond to actions in the underlying systems of asynchronously communicating EFSM's: they model the inputs and outputs, calculations in the context variables, and the setting, resetting and expiration of timers.

A *labeled event structure* over a set of actions A is a quadruple $\langle E, \preceq, \#, l \rangle$ where

- E is a finite set of *events*;
- $\preceq \subseteq E \times E$ is a partial order relation in E, called *causality relation*, such that for all $e \in E$ the set $\{e' \in E \mid e' \preceq e\}$ is finite (i.e., the number of causal predecessors of any event is finite);
- $\# \subseteq E \times E$ is an irreflexive and symmetric relation in E, called *conflict relation*, such that $\forall e, e', e'' \in E((e \# e' \wedge e' \preceq e'') \Rightarrow e \# e'')$ (i.e., conflicts are inherited: if an event e is in conflict to some event e', then it is also in conflict to all causal successors of e');
- $l : E \rightarrow A$ is a *labeling function* assigning an action to each event.

$e \preceq e'$ means that if the events e and e' both happen, then e must happen before e'. $e \# e'$ means that the events e and e' cannot happen both in a single run of the system. If two events are neither causally related nor in conflict, then they are *concurrent* to each other and both can occur in any order: either e before e', e and e' at the same time or e' before e. All events occurring in the same EFSM are either causally related or in conflict, but not concurrent to each other.

A labeled event structure is interpreted informally as follows: An event can occur if all its causal predecessors have occurred and no conflicting event has occurred yet.

Let $m_{les} = \langle E, \preceq, \#, l \rangle$ be a labeled event structure and $C \subseteq E$ be a subset of events of m_{les}. C is *causally closed* if $\forall e \in C \forall e' \in E (e' \preceq e \Rightarrow e' \in C)$. C is *conflict-free* if $\forall e, e' \in C (\neg (e \# e'))$. C is a *configuration* of m_{les} if it is causally closed and conflict-free. That means, a configuration is a set of events that have occurred by some stage in executing a labeled event structure. The *necessary configuration* $[e]$ of an event $e \in E$ of a labeled event structure m_{les} is the subset of events that includes *e* and all causal predecessors of *e*, but not any other events, i.e. $[e] = \{e' \in E \mid e' \preceq e\}$. All events that have to occur prior to an event *e* belong to the necessary configuration of *e*. Events that are concurrent to *e* do not belong to the necessary configuration of *e*.

Each configuration of a labeled event structure constructed from a system of asynchronously communicating EFSM's corresponds to a global state of the system. The *final state* $gs(C)$ of a configuration of a labeled event structure constructed from a system of asynchronously communicating EFSM's *m* is the global state of *m* reached after all events $e \in C$, but no other events have occurred.

The construction of a labeled event structure from a system of asynchronously communicating EFSM's can be cut off at different points, leading to different event structures. The labeled event structure obtained by unfolding a system of asynchronously communicating EFSM's as much as possible is referred to as *the* labeled event structure of the system. Only a complete prefix of the labeled event structure of a system of asynchronously communicating EFSM's is constructed in our approach. A *prefix* of the labeled event structure $\langle E, \preceq, \#, l \rangle$ is a labeled event structure $\langle E', \preceq', \#', l' \rangle$ induced by a causally closed subset of events $E' \subseteq E$. A prefix of the labeled event structure of a system of asynchronously communicating EFSM's is *complete* if it contains a configuration C for each reachable global state g of the system such that

- $g = gs(C)$, i.e., g is represented by C, and
- for each transition $g \xrightarrow{\mu / \omega} g'$ enabled in g with $\omega = \nu_1 \ldots \nu_p$, the prefix contains a configuration $C' = C \cup \{e, e_1, \ldots, e_p\}$ with $e, e_1, \ldots, e_p \notin C$ and $l(e) = \mu$, $l(e_1) = \nu_1$, …, $l(e_p) = \nu_p$.

A *maximal configuration* is a configuration to which no more events of the complete prefix of the labeled event structure can be added. An event *e* is a *maximal event* of a configuration *C* if there does not exist any $e' \in C$ with $e \preceq e'$.

Graphical Representation. A labeled event structure is represented as a graph where vertices represent events, directed edges lead to the immediate causal successors of an event, and undirected dashed edges connect events in immediate conflict. Next to an event *e* its label *l*(*e*) is indicated. The graph of a labeled event structure is cycle-free. The set of events occurring in the same EFSM induces a subgraph that is a directed tree. We draw the subgraphs for the parallel EFSM's with their edges in parallel.

Fig. 2 shows a complete prefix of the labeled event structure of the system of asynchronously communicating EFSM's in Fig. 1. The complete prefix is annotated with the global states at cut-off points and at recursion points. The prefix may be expanded by appending the sub-structures starting with the corresponding global states to the cut-off points.

Construction of a Labeled Event Structure. The algorithm for unfolding systems of asynchronously communicating state machines into labeled event structures resembles the reduced reachability analysis from [11, 12], yet the results are taken down in the form of event structures. These reduced reachability algorithms aim at alleviating the state explosion problem and yield reduced reachability trees whose nodes represent only certain reachable global states and whose directed edges represent sets of transitions concurrently executable in a certain global state. Intermediate global states reached while executing a set of concurrent transitions are not explicitly represented.

For finding cut-off points suitable for a complete prefix of the labeled event structure, [6] takes up an approach for coping with the state explosion problem in analyzing Petri nets with finite state space [13, 14]. The main idea can be outlined as fol-

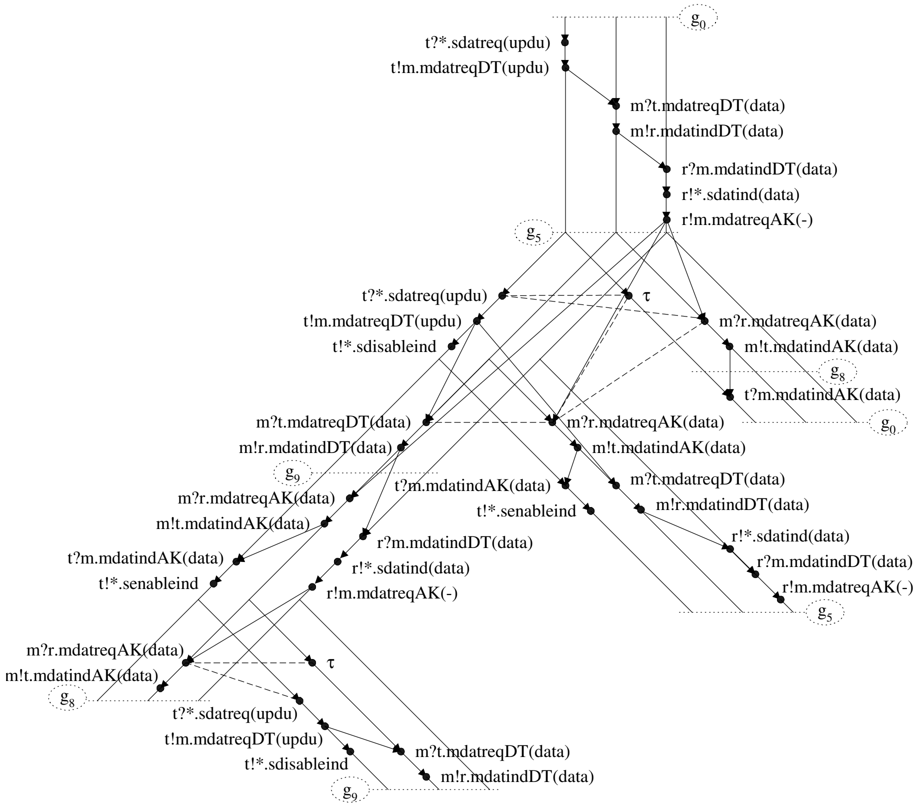

Fig. 2. A complete prefix of the labeled event structure for the example from Fig. 1

lows: An event is a cut-off event if its necessary configuration has the same final state as the necessary configuration of another event already contained in the unfolding. The unfolding can be cut off after these events since all events appended after the cut-off events would lead to states already covered by the prefix. [13] presents an algorithm for constructing a finite prefix of the unfolding of a Petri net. The prefix is complete with respect to the reachable markings of the Petri net. As the complete prefix of the unfolding constructed after [13] is sometimes larger than necessary, [14] improves the algorithm such that a complete prefix is constructed that is minimal in a certain sense. The algorithm in [14] is applicable to n-safe Petri nets with $n \geq 1$.

How a testing equivalent labeled event structure or its complete prefix can be constructed from a given system of asynchronously communicating state machines is treated in detail in [6]. The approach is applicable if all state machines of the system have a finite number of states and all queues of the system are bounded. This is not an undue restriction as in many cases an unbounded growth of the state space can be avoided by appropriate design criteria.

3 Test Generation Approach

3.1 Starting Point

Starting point for the generation of test purposes is a complete prefix m_{les} of the labeled event structure constructed from a system of asynchronously communicating EFSM's *m*. It forms a semantic model of a given specification of the implementation under test (IUT) embedded in a test context and hence models the behavior perceivable at the system boundaries during black box testing. The events that involve an interaction with the environment represent events occurring at points of control and observation (PCO's), i.e. at points where a test system may interact with IUT and test context.

As illustrated in Fig. 2, cut-off points and recursion points of m_{les} are labeled with the corresponding global states of the system *m* in order to characterize the possible continuations of the behavior.

3.2 Implicit Test Purposes and Test Coverage

As each maximal configuration of a complete prefix of the labeled event structure represents a significant behavior, it is desirable that a test suite tries to execute each maximal configuration of the complete prefix at least once. We also regard it as sufficient to execute each maximal configuration of the complete prefix once. This limits the size of the test suite. At the cut-off points of the complete prefix, behavior that has been encountered before is repeated anyway. By generating a larger test suite covering more than the complete prefix, one attains a higher test coverage and a higher degree of confidence that the IUT will operate free of error when actions are executed repeatedly. In principle, if the IUT is regarded as a black box, it remains uncertain whether or not it will operate free of error when the same actions are executed next time. Based on knowledge about the inner structure of an implementation (e.g. about the reliability of the operating system, about the programming language used, etc.), however, often it is inferred that an implementation will work free of error any number of times if it does so at least once.

For each maximal configuration of the complete prefix of the labeled event structure of a system of asynchronously communicating EFSM's a test case is to be generated. Its test purpose is to check the behavior described by the corresponding maximal configuration.

By covering each maximal configuration of the complete prefix, we achieve all-nodes coverage (or all-events coverage) w.r.t. the complete prefix. We do not necessarily achieve all-transition coverage w.r.t. the underlying system of asynchronously communicating EFSM's due to the fact that the EFSM's may contain transitions that are never triggered in normal interaction with the other EFSM's of the system.

3.3 Algorithm for Generating Test Purposes

Overview. The goal is to construct a set of test purpose descriptions in form of MSC's from the complete prefix m_{les} of the labeled event structure of a system of asynchronously communicating EFSM's. The generation of test purposes is carried out in the following steps, which are implemented as a prototype tool [15]:

1. Identify all maximal configurations of the complete prefix;
2. Restrict the maximal configurations to events occurring at the PCO's;
3. For each restricted maximal configuration, check whether it is included in another maximal configuration, and if so, eliminate it from the set of maximal configurations;
4. Format the maximal configurations as MSC's.

Identification of Maximal Configurations. In order to obtain the set of events belonging to a maximal configuration, we start from the cut-off points and follow the causality relation backwards to the roots. First, all the maximal events at a cut-off point are put into an initially empty event queue and into an initially empty event set. Loop while the event queue is not empty, get the first event from the queue and put all its predecessors that have not been put into the event set yet into the event queue and into the event set. When the loop terminates, all the events belonging to the maximal configuration have been put into the event set. After a maximal configuration is obtained, it is added to the set of maximal configurations.

The identification of all maximal configurations is described in pseudo-code below. $mconf_i$ denotes a maximal configuration from the set of all maximal configurations $MCONF$. $cutoff_i$ denotes the set of maximal events at a cut-off point. $m_{les}.CUTOFF$ denotes the set of all cut-off points of m_{les}. $pred\_queue$ is the queue data structure for processing the predecessor events.

```
$MCONF := \emptyset$;
for all $cutoff_i \in m_{les}.CUTOFF$ do
    $mconf_i := \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$;
    $pred\_queue := \emptyset$;
    for all $e_j \in cutoff_i$ do
        $mconf_i.E := mconf_i.E \cup e_j$;
        $put(pred\_queue, e_j)$;
    endfor;
    while not $empty(pred\_queue)$ do
        $ev := get(pred\_queue)$;
        for all $e_j \in ev.predecessors$ do
            if $e_j \notin mconf_i.E$ then
                $mconf_i.E := mconf_i.E \cup e_j$;
                $put(pred\_queue, e_j)$;
            endif;
        endfor;
    endwhile;
    $MCONF := MCONF \cup mconf_i$;
endfor;
```

Restriction to Events at PCO's. The restriction has to be done because only the events occurring at the system boundaries can be controlled or observed during black box testing.

The process of restricting a maximal configuration to events occurring at the PCO's consists of checking all events in the maximal configuration and omitting the events for which the remote communication partner is not the environment. In restricting the maximal configurations, the transitivity of the causality relation has to be preserved.

Below, the restriction to events occurring at PCO's is described in pseudo-code.

```
for all $mconf_i \in MCONF$ do
    for all $e_j \in mconf_i.E$ do
        if $(l(e_j).rem \neq$ "*") then
            $mconf_i.E := mconf_i.E \setminus \{e_j\}$;
        endif;
    endfor;
endfor;
```

Inclusion Checking. In order to get a minimal set of maximal configurations, each configuration is checked, after restricting it to the events occurring at the PCO's, whether it is included in another configuration in the obtained set of restricted maximal configurations. If so, it is removed from the set.

Below is pseudo-code for the inclusion checking.

```
for all $mconf_i \in MCONF$ do
    for all $mconf_j \in MCONF$ $(i \neq j)$ do
        if $mconf_i.E \subseteq mconf_j.E$ then
            $MCONF := MCONF \setminus mconf_i$;
        endif;
    endfor;
endfor;
```

Formatting Maximal Configurations as MSC's. The test purpose descriptions can be laid out as process-level MSC's or as system-level MSC's.

A maximal configuration of a complete prefix of the labeled event structure of a system of asynchronously communicating EFSM's can be straightforwardly interpreted as a process-level MSC with one instance for every EFSM associated with a PCO and one instance for every PCO. This way, the concurrency of different EFSM's remains unhidden. Fig. 3 shows a test purpose description for the example in Fig. 1 in form of a process-level MSC. The interfaces to the environment on transmitter and receiver side are referred to as PCO_t and PCO_r, respectively.

On the other hand, test purpose descriptions for Autolink are stored as system-level MSC's containing only one instance for the whole system under test and one instance for every PCO [4]. This way, the concurrency of different components of the system is hidden. To make the output of our tool applicable as input to Autolink, our tool also generates system-level MSC's.

To generate system-level MSC's, we have to linearize the maximal configurations. A linearization of a partially ordered event set is a total order on this event set that contains the partial order. A linearization can be derived from a configuration by adding arbitrary ordering constraints to the partial order of the configuration.

In order to get a linearization for a maximal configuration restricted to the PCO's, first, all the events in the maximal configuration are put into an initially empty event queue. Loop while the event queue is not empty, get the first event from the queue, check whether all its predecessor events are already included in the linearization. If so, add the event to the linearization. The first event added to the linearization will be an initial event without any predecessor. If not yet all predecessor events are in the linearization, put the event again into the event queue.

The linearization of maximal configurations is described in pseudo-code below. $mconf_i.seq$ denotes the linearization of a maximal configuration. *e_queue* is the queue data structure for processing the events.

```
for all $mconf_i \in MCONF$ do
    $mconf_i.seq := \emptyset$;
    $e\_queue := \emptyset$;
    for all $e_j \in mconf_i.E$ do
        $put(e\_queue, e_j)$;
    endfor;
    while not $empty(e\_queue)$ do
        $ev := get(e\_queue)$;
        if $(ev.predecessors \subseteq mconf_i.seq)$ then
            $mconf_i.seq := concatenate(mconf_i.seq, ev)$;
        else $put(e\_queue, ev)$;
        endif;
    endwhile;
endfor;
```

Fig. 4 shows a test purpose description for the example in Fig. 1 in form of a system-level MSC.

3.4 Data Flow Aspects

The complete prefix of the labeled event structure constructed from a set of asynchronously communicating EFSM's without enabling conditions for transitions may contain variables that are used for representing message parameters, for buffering values, or for calculations to be carried out during the execution of transitions. It does not contain enabling conditions for the occurrence of events. Therefore, the occurrence of each configuration in the complete prefix is feasible.

Some data flow oriented test selection criteria that have been introduced for specifications represented by directed graphs can be transferred to labeled event structures. These criteria establish associations between definitions and uses of variables. Such associations are identified by tracking variables through the specification, following them as they are modified, until they are ultimately used in outputs or to compute values for other variables. The criteria require that each of these associations be examined at least once during testing. The intuition behind the selection of tests based on the coverage of data flow associations is that faults in a system may lead to incorrect values and, as a result of propagation through computations, an error may show up at the system's output.

The all-uses coverage criterion is satisfied w.r.t. the complete prefix of the labeled event structure if for each variable defined in the complete prefix each subsequent use of that variable (i.e., each def-use pair) is covered by at least one test. Even if there are no definitions without subsequent use within the underlying system of asynchronously communicating EFSM's, not necessarily all variables defined within the complete prefix of the labeled event structure are used within the complete prefix. To achieve full all-uses coverage, our tool appends sub-structures of the complete prefix to the cut-off points whenever necessary and possible for covering definitions without use within the complete prefix.

4 Summary and Outlook

The approach introduced in this paper generates test purpose descriptions in form of MSC's from a non-interleaving model, viz. from a complete prefix of the labeled event structure constructed from a system of asynchronously communicating EFSM's.

This model alleviates the state-explosion problem and preserves true concurrency. The size of the resulting test suite is restricted in a suitable way. The approach is applicable to a large class of specifications. The executability of the test cases is ensured.

A prototype tool implementing the approach described in this paper is available [15]. Its input is generated by the prototype tool for constructing a complete prefix of the labeled event structure from a generalized model of asynchronously communicating state machines [6]. Together with the corresponding system specification, the output of the test purpose tool is intended as input for test generation tools that take explicit descriptions of test purposes as input.

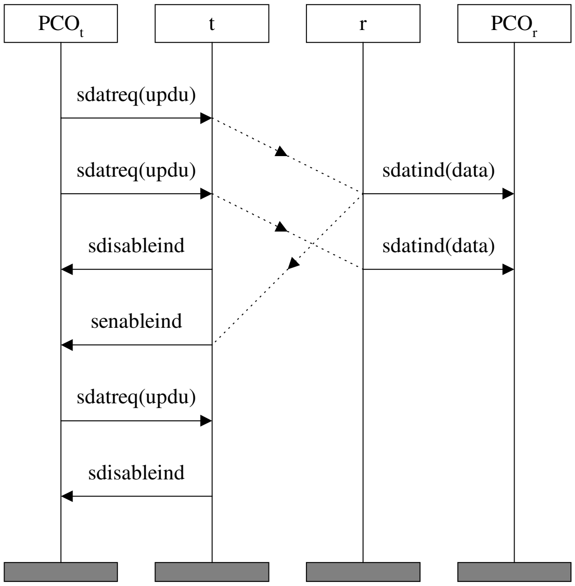

Fig. 3. A test purpose description as process-level MSC

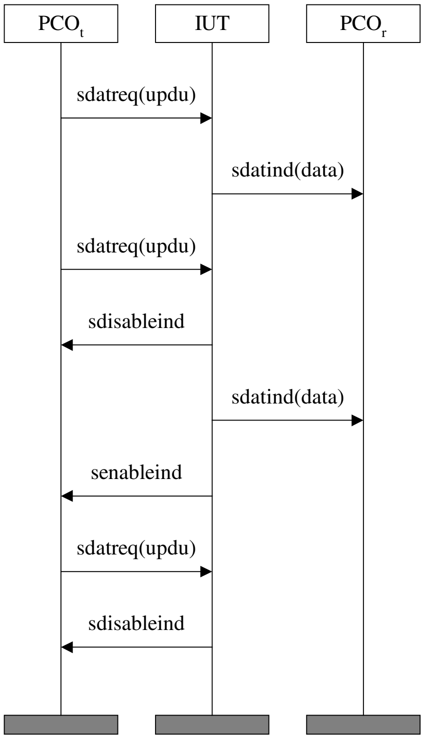

Fig. 4. A test purpose description as system-level MSC

As an alternative to the textual and tabular presentation formats, the new version of TTCN (TTCN-3) [16] allows describing tests in a graphical presentation format based on a subset of MSC's. The MSC's generated by our tool describe only the desired behavior to be checked in a test case. Therefore, the generated MSC's are used as test purpose descriptions. MSC's for defining test cases have to describe the behavior of the test components interacting with IUT and test context via the PCO's and to cover possible behavior alternatives, which would lead to inconclusive or fail verdicts. The verdicts have to be included in a test case as well. The direct generation of MSC test cases from a labeled event structure is an area of future work.

Acknowledgements

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada.

References

1. Specification and Description Language (SDL-2000), ITU-T Recommendation Z.100, 1999
2. Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 1: General concepts, International Standard ISO/IEC 9646-1, 2nd edition, 1994
3. Message Sequence Chart (MSC), ITU-T Recommendation Z.120, 1999
4. B. Koch, J. Grabowski, D. Hogrefe, M. Schmitt, "Autolink – a tool for automatic test generation from SDL specifications", in Proc. of the 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques, Boca Raton, Florida, USA, 1998
5. Kerbrat, T. Jéron, and R. Groz, "Automated test generation from SDL specifications", in [17], pp. 135–151
6. O. Henniger, "Test generation from specifications in Estelle and SDL", Ph.D. thesis (in German), Brandenburg Technical University of Cottbus, Germany, 2002, Shaker Verlag
7. J. Grabowski, B. Koch, M. Schmitt, D. Hogrefe, "SDL and MSC based test generation for distributed test architectures", in [17], pp. 389–404
8. C. Jard, "Synthesis of distributed testers from true-concurrency models of reactive systems", *Information and Software Technology* 45, pp. 805–814, 2003
9. O. Henniger, A. Ulrich, H. König, "Transformation of Estelle modules aiming at test case generation", in A. Cavalli, and S. Budkowski (eds.), Proc. of IWPTS'95, Evry, France, 1995, pp. 36–51
10. M. Nielsen, G. Plotkin, and G. Winskel, "Petri nets, event structures and domains, Part I", *Theoretical Computer Science* 13, 1981, pp. 85–108
11. M. Itoh and H. Ichikawa, "Protocol verification algorithm using reduced reachability analysis", *Transactions of the IECE of Japan* E 66 (2), pp. 88–93, 1983
12. N. Arakawa and T. Soneoka, "A test case generation method for concurrent programs", in J. Kroon, R.J. Heijink, E. Brinksma (eds.), Proc. of IWPTS'91, Leidschendam, The Netherlands, 1991, pp. 95–106

13. K.L. McMillan, “A technique of state space search based on unfolding”, *Formal Methods in System Design* 6 (1), 1995
14. J. Esparza, S. Römer, and W. Vogler, “An improvement of McMillan’s unfolding algorithm”, *Formal Methods in System Design* 20 (3), pp. 285–310, 2002
15. M. Lu, “Generation of tests from labeled event structures”, M.Sc. thesis, University of Ottawa, Canada, May 2003
16. Methods for Testing and Specification; The Testing and Test Control Notation Version 3; Part 3: TTCN-3 Graphical Presentation Format (GFT), ETSI Standard ETSI ES 201 873-3, February 2003
17. R. Dssouli, G.v. Bochmann, and Y. Lahav (eds.), Proc. of the 9th SDL Forum, Montréal, Québec, Canada, 1999