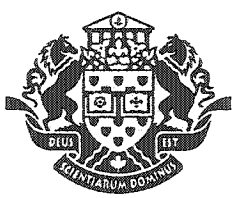


Université d'Ottawa · University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Lu, Miao

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

Master of Computer Science

GRADE - DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

Generation of Tests from Labeled Event Structures

H. Ural

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

CO-DIRECTEUR DE LA THÈSE - THESIS CO-SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

F. Bordeleau

A. Williams

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

SIGNATURE

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

GENERATION OF TESTS FROM LABELED EVENT STRUCTURES

by

Miao Lu

A M. Sc. Thesis

Submitted to the School of Graduate Studies and Research
Of the University of Ottawa
In partial fulfillment of the requirement for the degree of

Master of Computer Science*

Department of Computer Science
University of Ottawa
Ottawa, Ontario

* The Master of Computer Science program is a joint program with Carleton University, administrated by the Ottawa-Carleton Institute for Computer Science

Your file Votre référence
ISBN: 0-612-89898-9
Our file Notre référence
ISBN: 0-612-89898-9

Abstract

Control and data flow aspects of systems of communicating processes can be identified through the analysis of control and data dependencies that exist not only within processes, but also across process boundaries. The existing methods for test generation for distributed systems can be roughly classified into methods with explicit test purposes and methods with implicit test purposes: methods with explicit test purposes allow the test designer to choose what to test but require considerable manual effort and do not guarantee systematic test coverage; whereas most of the existing methods with implicit test purposes are either restricted to a small subset of systems usually consisting of a single process, or suffer from the state-explosion problem.

In this thesis, we present an algorithm for generating test cases for both control flow and data flow oriented testing from a given labeled event structure. A labeled event structure represents the behavior of a distributed system given as a system of asynchronously communicating EFSMs. Labeled event structure preserves the independency of events occurring at different EFSMs and reduces the combinatorial explosion in the number of possible event sequences (interleavings). The algorithm to generate a non-interleaving model of labeled event structure from a generalized model of asynchronously communicating EFSMs is given in [7]. Our system increases the input domain of the specifications of EFSMs by supporting actions of input, output, assignment, set, reset, timeout and procedure call. The set of tests for control flow oriented testing generated by our system satisfies all-nodes test selection criterion and the set of tests for data flow oriented testing generated by our system satisfies all-uses test selection criterion. All tests constructed by our system are feasible because of the absence of path predicates in the labeled event structure.

Our system takes in a labeled event structure, converts the information of the labeled event structure into a set of defined internal data structures. Then, by analyzing control and data dependencies of the labeled event structure, two sets of tests (one set for control flow oriented testing and the other set for data flow oriented testing) are obtained.

Finally, all the tests are output in textual MSC format. Implementation details of a prototype are discussed and examples produced by the prototype are provided.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Hasan Ural, for all the guidance and valuable advice he has given me throughout my graduate studies at University of Ottawa. His dedication to his work is always going to inspire me.

I would like to sincerely thank Dr. Olaf Henniger for the valuable advice he has given me during the research of this thesis. I especially wish to thank him for the time he devoted and patience he showed when introducing the labeled event structure.

I would like to thank the (ASERT) Advanced Software Engineering Research and Training Group, in particular, to Dr. Robert Probert for providing an excellent research environment for the entire period of my study leading to this thesis.

I acknowledge my colleagues of ASERT, especially Craig Williams, Ruoshan Guan and Yiqun Xu for their valuable discussions and suggestions.

I especially wish to thank my parents, Jingyang Lu and Guiju Yue, and my sisters, Ying Lu and Ying Lu, for their continuous encouragement and support throughout my graduate studies which leads to the completeness of the thesis. I would like to thank my friend, Meng Yu, for his support and help during my studies.

I also like to acknowledge the Natural Sciences and Engineering Research Council for financially supporting this research and the School of Information Technology and Engineering of the University of Ottawa for providing me Teaching Assistantship.

Chapter 1 Introduction

1.1 Background

Software testing is one of the most widely used methods for software quality assurance. The general goal of software testing is to reveal the existence of errors in a program through the application of a *test suite* (i.e. a finite set of test cases, each consisting of *test input* and corresponding *expected output*) in a controlled environment. Generally, software testing methods can be categorized into three classes: *black box testing* (also called *functional testing*), *grey box testing*, and *white box testing* (also called *structural testing*), depending on the resource used for the construction of test cases. Test cases are derived from specification, design, source code of the program under test in black, grey, or white box testing, respectively [18, 31]. This thesis deals with black-box testing and in particular, test generation from specifications of concurrent systems.

The behavior of concurrent systems can be modeled by means of a set of extended finite state machines (EFSMs) that run concurrently and communicate with each other via First-In-First-Out (FIFO) queues. The formal description techniques Estelle [24] and SDL [25] are based on such a model, extended by additional features.

The behavior described by an individual state machine is characterized by a set of sequences (i.e. totally ordered sets) of events. The behavior described by a set of communicating state machines could also be characterized by the set of all possible sequences of events, where events of the individual state machines are intermixed. In fact, this is done in approaches generating a composite state machine and in interleaving semantics definitions.

Interleaving models preserve many properties of the original specification and are relatively easy to formalize. However, the combinatorial explosion in the number of possible event sequences (interleavings) forms a major problem in generating interleaving models and renders these approaches infeasible in many practical cases. Knowledge of all possible interleavings is in many cases not necessary: If events

occurring at different state machines are independent, then their particular order of occurrences does not change their combined effect [6].

Furthermore, interleaving models hide the independence of events occurring in different state machines. This is a problem in test case generation where often a finite subset of possible event sequences has to be selected: In case of independent events from different state machines, there is no way to control the order of occurrence of these events, and there is no guarantee that a selected interleaving can really be observed.

Concurrent TTCN is an amendment to the test description language TTCN [11] designed to specify test cases in a multi-party testing context. In conventional TTCN, the behavior descriptions of all test components have to be interleaved in a single tree. Even if split up into several local trees or test steps attached to each other, the behavior description of a test case forms a single "evaluation tree" as defined in the operational semantics of conventional TTCN. Concurrent TTCN allows for independently executable test components each of which processes its own evaluation tree [26].

The existing methods for test generation from specifications of concurrent systems can be roughly classified into methods with explicit test purposes and methods with implicit test purposes.

The methods with explicit test purposes require the test designer to choose what to test and ensure that test cases consistent with the specification and the test purposes are generated. These methods offer much flexibility, but require considerable manual effort and do not guarantee a systematic test coverage [14, 15, 16, 17, 29].

The methods with implicit test purposes offer less flexibility in choosing for which faults to generate test cases, but they require less manual efforts and guarantee a systematic test coverage. Some of these methods focus on the construction of test sequences for testing the control flow aspects [19, 20]; other methods focus on the construction of test sequences for testing the data flow aspects [2, 3, 5, 21, 23]. Since these methods consider only a single EFSM and a limited syntax for the EFSM representation, their applicability is restricted to a small subset of concurrent systems.

Some other methods have been proposed for systems of communicating EFSM's. As they need to explore the possible behavior of the system, they suffer from the state-explosion problem [7, 34, 35, 36].

In [6], an approach for generating test cases in Concurrent TTCN from a system of asynchronously communicating finite state machines with implicit test purposes has been proposed, an algorithm for transforming a system of asynchronously communicating state machines into a Labeled Event Structure (LES) (a non-interleaving behavior model of concurrent processes) is given, and a prototype tool for constructing a labeled event structure is implemented. This thesis complements the approach in [6] by targeting automatic generation of test cases for both the control flow aspects and the data flow aspects of a labeled event structure.

1.2 Contributions of the Thesis

A method for generating test cases from a given labeled event structure is studied in this thesis. An algorithm for constructing test cases from the labeled event structure is developed and a prototype tool for the test case generation is implemented in C++. The procedure of the test case generation takes in a labeled event structure and outputs Message Sequence Charts (MSC) that represent the tests covering control or data flow aspects of the labeled event structure. This procedure is divided into three distinct phases:

- Processing a labeled event structure;
- Configuration construction;
- Test construction.

Compared with other related work, our approach has the following advantages:

- Automation of both control and data flow oriented test suite construction is achieved.
- All the tests generated are feasible, and thus no postprocessing to determine whether a generated test is feasible is needed.

1.3 Organization of the Thesis

The rest of this thesis is organized as follows: Chapter 2 presents the labeled event structures, reviews test generation methods, and introduces concepts of control flow and data flow oriented testing. The proposed test suite generation process is discussed in Chapter 3. First, the translation from the labeled event structure to an internal data structure for the labeled event structure is given (Phase 1). Then, an algorithm for the configuration generation for both control flow and data flow oriented testing is presented (Phase 2). Finally, the details of test data selection are given (Phase 3). A prototype of the proposed process is presented and some examples are analyzed in Chapter 4. Comparison with other related systems, concluding remarks and future research directions are given in Chapter 5.

Chapter 2 Labeled Event Structures and Test Derivation Methods

2.1 Labeled Event Structures

2.1.1 A Model for Concurrency

Various formalisms have been defined for describing the behavior of concurrent systems. Models for concurrency can be classified into [1]:

- behavior or system models,
- interleaving or non-interleaving models,
- linear-time or branching-time models.

Behavior models describe the behavior in terms of the order of events, abstracts away from states, while system models describe the order of events implicitly and represent states, which are repeatable, explicitly. Interleaving models hide the difference between concurrency between several state machines and non-determinism inside individual state machines, while non-interleaving models take the difference into account. Branching-time models represent the branching structure of the behaviors, show out the points which choices are taken, while linear-time models do not show the branching structure. [7]

The labeled event structure used in this thesis can be characterized as a behavior, non-interleaving, and branching-time model, which will be used to represent the behavior of a system of asynchronously communicating EFSMs. We consider only EFSMs without enabling conditions for transitions. In these EFSMs, variables are used only for calculations to be carried out during the execution of transitions and for representing input and output parameters, but not for specifying enabling conditions for transitions. 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 [3, 6].

Let $m = \langle M, Q \rangle$ be a system of asynchronously communicating EFSMs with $M = \{m_1, \ldots, m_n\}$, which represents a set of asynchronously communicating processes, and $Q = \{q_1, \ldots, q_r\}$, which represents a set of message queues between the communicating processes.

Global state of m can be represented as $g = (s_{m1}, \ldots, s_{mn}, c_{q1}, \ldots, c_{qr})$, where

- $s_{m1} \ldots s_{mn}$ are the current states of the asynchronously communicating EFSMs $m_1 \ldots m_n$ and
- $c_{q1} \ldots c_{qr}$ are the contents of the queues between the EFSMs.

A Labeled Event Structure (LES) σ, 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 relationship, 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, 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 with some event e', then it is also in conflict with 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' can't 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 arbitrary order. All events occurring in the same EFSM are either causally related or in conflict, but are not concurrent to each other.

A basic element of labeled event structures is 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 considered in this thesis correspond to the actions in the underlying systems of communicating EFSMs: they model the sending and receiving of messages, calculations in the variables of EFSMs, and the setting, resetting and expiring of timers.

Let $\sigma = \langle E, \preceq, \#, l \rangle$ be a labeled event structure and $C \subseteq E$. C is *causally closed* iff $\forall e \in C \forall e' \in E(e' \preceq e \Rightarrow e' \in C)$. C is *conflict-free* iff $\forall e, e' \in C(\neg(e \# e'))$. C is a configuration of σ iff it is causally closed and conflict-free. A *configuration* is a set of events that have occurred by some stage in a labeled event structure. $C(\sigma)$ denotes the set of all finite configurations of σ.

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

Without being cutoff, the labeled event structure corresponding to a system of asynchronously communicating EFSMs can be constructed infinitely, unless there are states with no exit ("sink" states). But, infinite is typical. 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 EFSMs 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 $\omega = v_1 ... v_p$, the prefix contains a configuration $C' = C \cup \{e, e_1, ..., e_p\}$ with $e, e_1, ..., e_p \notin C$ and $l(e) = \mu$, $l(e_1) = v_1, ..., l(e_p) = v_p$.

The *necessary configuration* $[e]$ of an event $e \in E$ of a labeled event structure σ 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.

A *maximal configuration* is a configuration to which no more events of the complete prefix of the labeled event structure can be added.

Figure 2.1 shows an example of a system of asynchronously communicating EFSM's.

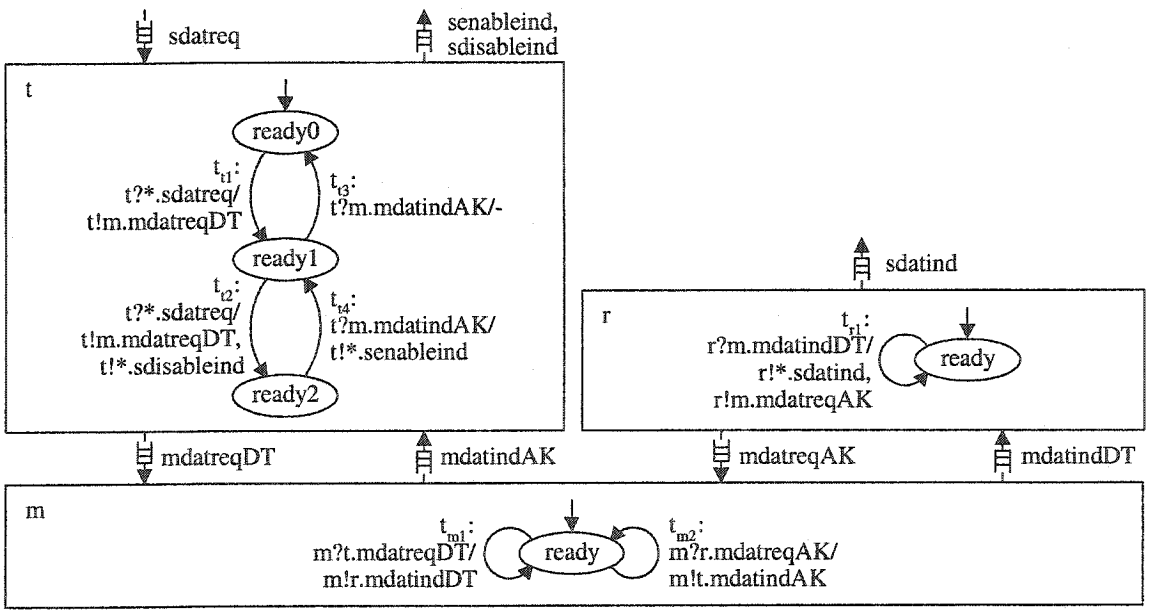


Figure 2.1 Example 1, A System of Asynchronously Communicating EFSMs

There are three components in the system of asynchronously communicating EFSM's in Figure 2.1: component "t", "m" and "r". Signals are exchanged between two

components or one component and the test environment through signal stacks. "*" is used as the remote component name for signals that are received from or sent to the test environment. Inside each component, the ovals indicate the states of the component. The directed arcs indicate the transitions between the states of the component. For example, the transition in component "r" is: "t_{r1}: r?m.mdatindDT / r!*.sdatind, r!m.mdatreqAK", which means that at state "ready" component "r" can receive signal "mdatindDT" from component "m", and then send out signal "sdatind" to the test environment and signal "mdatreqAK" to component "m". In our approach, we assume that if there exists a comma between two actions of one transition, the action before the comma must occur before the action after the comma.

The functionality of the system in Figure 2.1 is to transmit data from the transmitter (component "t") to the receiver (component "r") and to protect the receiver against overloading. If the number of messages for which the acknowledgement is outstanding reaches its maximum (in our case for simplicity, it is "2"), the transmitter outputs "sdisableind" to indicate that for the time being no more messages can be transmitted. When the transmitter receives an acknowledgement afterwards, then "senableind" is output and new messages can be transmitted again.

Figure 2.2 shows a complete prefix of the labeled event structure for the Example 1 of Figure 2.1. As shown in Figure 2.2, a labeled event structure is represented as a graph where bold-faced points represent events, directed arcs lead to the immediate causal successors of an event, and undirected dashed arcs connect events in immediate conflict. Next to an event e, its label $l(e)$ is indicated. The vertical dashed lines indicated at the margin, marked by letter "g" followed by a number, represent the global states. They function as labels indicating where behavior encountered earlier repeats. Events occurring in the same EFSM form a directed tree. The trees for the communicating EFSMs are drawn with parallel arcs.

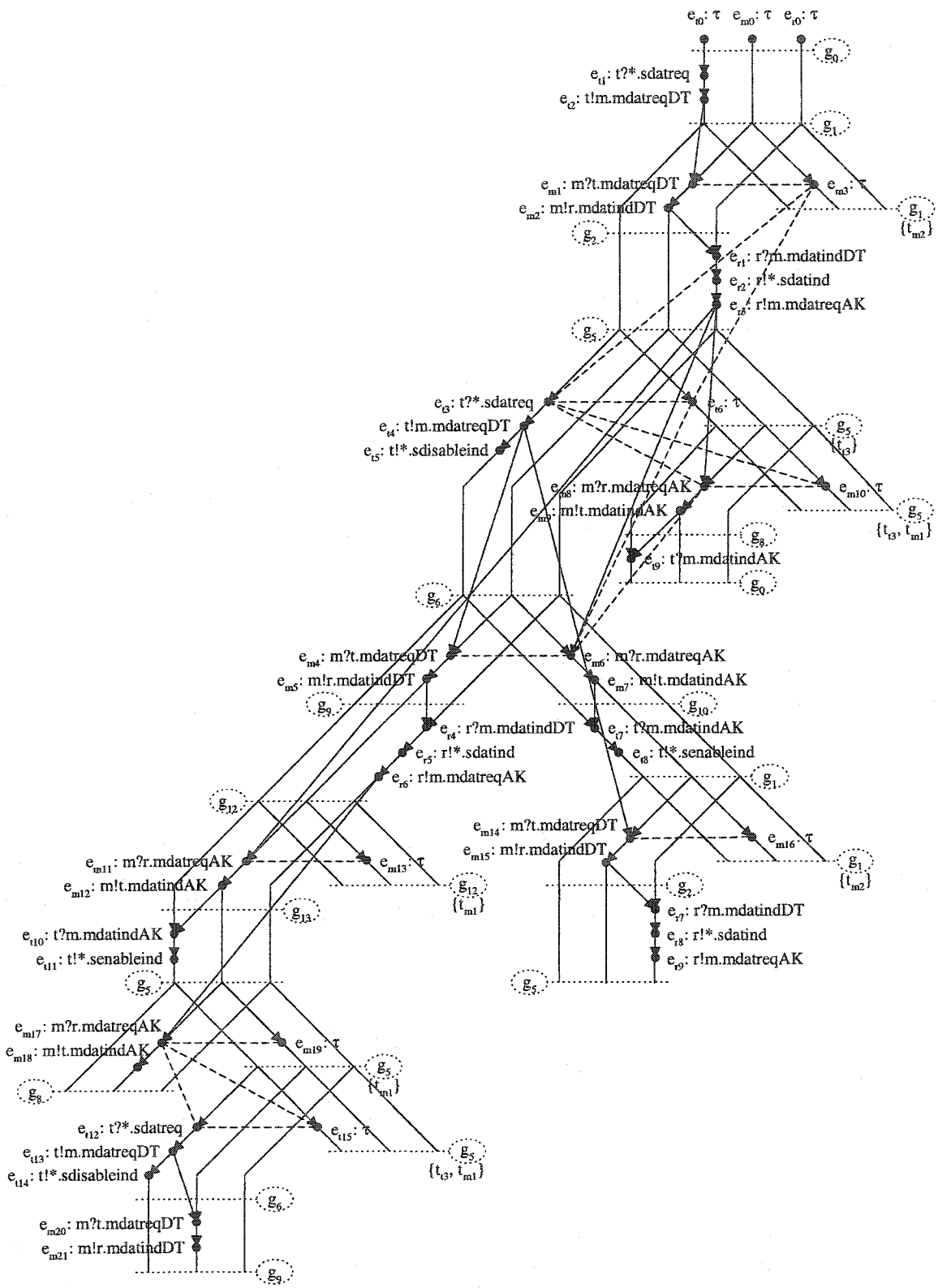


Figure 2.2 Labeled Event Structure for Example 1

The labeled event structure in Figure 2.2 is constructed by three parallel trees corresponding to the three components in Figure 2.1: t, m and r. The label of each event contains the component name indicating where it occurs, a number indicating the occurrence order of that particular event in that component, and the action of the event. For example, from the label of the third event at component "t", which is "e_{t3}:t?*.sdatreq", we can tell that this event is the third event occurring at component "t", its action is to receive signal "sdatreq" from the test environment. Also from the graph we can tell that event "e_{t3}" is in causality relation with event "e_{t4}" and in conflict relation with event "e_{t6}". It has neither causality relation or conflict relation with event "e_{r3}", so they are concurrent events, their occurrence order in the test will not affect the test result.

2.1.2 Construction of Labeled Event Structures

The starting point for the process of constructing a labeled event structure is a system of asynchronously communicating EFSMs as specification of the implementation under test (IUT). To describe the algorithm for constructing a complete prefix of the labeled event structure of a system of asynchronously communicating EFSMs, we need the following definitions:

An *enabled transition* of an EFSM $m_k \in M$ in the global state g of $m = \langle M, Q \rangle$ is a transition $s \xrightarrow{\mu / \omega} s'$ for which the current state of m_k is s and μ is either

- a trigger input action from environment m_k?*.*msg* ("*" stands for the environment);
- the non-observable input action τ ;
- an input action m_k?*rem.msg* (*rem* $\neq$ *) and the message *msg* is at the head of the queue q_i with $destination_{q_i} = m_k$ and $rem \in source_{q_i}$, i.e. $front(c_{q_i}) = msg$, where *rem* denotes remote component. For an input action (e.g., m_k?*rem.msg*), remote component is the component which sends out the signal; for an output action, (e.g., m_k!*rem.msg*), remote component is the component which receives the signal.

$ET_{mk}(g)$ denotes the set of all enabled transitions of m_k in the global state g of m.

A *potentially enabled transition* of an EFSM $m_k \in M$ in the global state g of $m = \langle M, Q \rangle$ is a transition $s \xrightarrow{\mu / \omega} s'$ for which the current state of m_k is s and μ is an input action $m_k?rem.msg$ ($rem \neq *$), but the queue q_t with $destination_{qt} = m_k$ and $rem \in source_{qt}$ is empty.

$PT_{mk}(g)$ denotes the set of all potentially enabled transitions of m_k in the global state g of m.

In order not to lose information about possible continuations of the behavior while translating a system of asynchronously communicating EFSMs into a labeled event structure, the algorithm given in [6] keeps track of not only the labeled event structure, but also the global state reached after the constructed configurations (the final state of the configuration), to which further events can be appended.

Inside the data structure describing a configuration, the following fields are needed for the algorithm:

- gs, the final state of the configuration,
- $predecessor_{mi}$ $(1 \leq i \leq n)$, the last event for each FSM m_i,
- $send_v$ for each output action v. $send_v$ is a FIFO queue data structure for storing the events labeled with the output action v in the order of their occurrence. The events in $send_v$ are assigned in the order of their occurrence as causal predecessors to the complementary input events (which receive the message sent by v).
- $Wait_{mi}$ $(1 \leq i \leq n)$, indicates the wait set of component m_i of a labeled event structure. $Wait_{mi}$ is a set of potentially enabled transitions of EFSM m_i that m_i "waits" for. If $Wait_{mi}$ is not empty, no transition of m_i can be appended to the labeled event structure unless one of the potentially enabled transitions in $Wait_{mi}$ becomes enabled. As long as no transition in $Wait_{mi}$ becomes enabled, we can only

append executable transitions of some other EFSM m_j to the labeled event structure.

The final state of a configuration can be obtained step by step starting from the initial configuration and the initial global state. Let $g = (s_{m1}, \ldots, s_{mn}, c_{q1}, \ldots, c_{qr})$ be a global state, $t = s \xrightarrow{\mu / \omega} s'$ with $\omega = m_k!rem_1.msg_1, \ldots, m_k!rem_p.msg_p$ be an enabled transition in g of an EFSM $m_k \in M$. next(g, t) is the global state reached from global state g by transition t and is defined as follows:

- If μ is a trigger input action $m_k?*.msg$ or the non-observable input action τ and the state of m_k is s in the global state g, i.e. $s_{mk} = s$, then next (g, t) is the global state $(s'_{m1}, \ldots, s'_{mn}, c'_{q1}, \ldots, c'_{qr})$ with

  - s'_{mi} is s' when $i = k$ and s_{mi} when $i \neq k$, $1 \leq i \leq n$;

  - c'_{qj} is enqueue(c_{qj}, msg_h), when $destination_{qj} = rem_h$ ($1 \leq h \leq p$) and $m_k \in source_{qj}$, $1 \leq j \leq r$ and c_{qj} , $1 \leq j \leq r$, in all other cases.

- If μ is an input action $m_k?rem.msg$ ($rem \neq *$), the state of m_k is s in the global state g, i.e. $s_{mk} = s$, and the message msg is at the head of the queue q_t with $destination_{qt} = m_k$ and $rem \in source_{qt}$, i.e. $front(c_{qt}) = msg$, then $next(g, t)$ is the global state $(s'_{m1}, \ldots, s'_{mn}, c'_{q1}, \ldots, c'_{qr})$ with

  - s'_{mi} is s' when $i = k$ and s_{mi} when $i \neq k$, $1 \leq i \leq n$;

  - c'_{qj} is dequeue(c_{qj}), when $destination_{qj} = m_k$, $rem \in source_{qj}$, front(c_{qj}) = msg, $1 \leq j \leq r$; or enqueue(c_{qj}, msg_h), when $destination_{qj} = rem_h$ ($1 \leq h \leq p$) and $m_k \in source_{qj}$, $1 \leq j \leq r$ and c_{qj} , $1 \leq j \leq r$, in all other cases.

The algorithm in [6] for constructing a labeled event structure from a system of asynchronously communicating EFSMs is outlined as follows:

It starts with the initial global state and the initial configuration. For each EFSM $m_i \in M$, the initial configuration contains an event e_{mi0} labeled with the non-observable

action τ . The final state of the initial configuration is the initial global state. At the beginning of the appending procedure, new events are appended to the labeled event structure from the initial global state, by which new global states are obtained. As the appending procedure going on, events are appended to those new global states.

The rules for appending events to the labeled event structure from a global state can be described as follows:

- Firstly, check if there exists an EFSM in the system that has only one enabled transition in the final state of the current configuration and there is no potentially enabled transition in this EFSM, then the enabled transition is appended as linear continuation to the current configuration. If there is more than one EFSM satisfying this condition, only one EFSM is chosen arbitrarily.

- If there is no such EFSM existing in the system (in other words, for each EFSM of the system, there are more than one enabled or potentially enabled transitions existing), one EFSM should be chosen arbitrarily and the labeled event structure has to be branched out. The arbitrary choice of the EFSM does not affect the constructing of the labeled event structure, because whichever EFSM is chosen, a transition that is enabled in other EFSMs remain enabled, no matter whether meanwhile a transition of the chosen EFSM is executed. Thus, in a concurrent model, the different execution orders for each EFSM, that can be appended at the same point, do not change the result of test set. At the branch out point, for an EFSM m_i, all enabled alternative transitions are appended to the current configuration, by opening up new branches. If the chosen EFSM m_i has potentially enabled transitions in the final state of the current configuration, an additional new branch is opened up with a non-observable action τ , and the potentially enabled transitions are passed to the wait set of the EFSM m_i of the additional new branch of the current configuration. The events of each of the new branches are in conflict to the events which are appended to other branches.

Among the rules for appending, the linear appending has the highest priority. Appending a linear continuation before possible branching continuations can avoid

appending the same event to each branch separately, thus the state space of the labeled event structure is not unnecessarily enlarged.

The appending of events to a labeled event structure cannot be performed endlessly. The termination condition of the construction is the cutoff events [13], after which the construction can be stopped without losing information about the reachable global states of the system. An event is a *cutoff event* if its necessary configuration has the same final state as the necessary configuration of another event already contained in a prefix of the labeled event structure.

Let A be the set of all observable actions of a system. Let $<_A$ be an arbitrary, linear order relation in $A \cup \{\tau\}$. If C is a set of events labeled with actions from $A \cup \{\tau\}$, then $\varphi(C)$ is the sequence of these events ordered according to $<_A$.

Let C_1 and C_2 be two configurations of a labeled event structure.

$\varphi(C_1) <_A \varphi(C_2)$ if $\varphi(C_1)$ is lexicographically smaller than $\varphi(C_2)$ with respect to $<_A$.

$C_1 \prec C_2$ if

- $|C_1| < |C_2|$ or
- $|C_1| = |C_2|$ and $\varphi(C_1) <_A \varphi(C_2)$.

The order relation $\prec$ [13] in the set of finite configurations of a labeled event structure is to describe formally that a configuration is "smaller" than another configuration or that a configuration has been reached "before" the other one during the construction. If the larger one of two configurations has the same final state as the smaller one, then the construction of the prefix can be stopped after the larger one.

An event $e \in E$ is a cutoff event of a prefix of the labeled event structure of a system of asynchronously communicating EFSMs if E contains an event e' with a

necessary configuration [e'] that is smaller than [e] and that has the same final state as [e], i.e. $\exists e' \in E(([e'] \prec [e]) \wedge (gs([e']) = gs([e])) \wedge (Wait([e']) = Wait([e])))$.

Thus, a set of cutoff events of a labeled event structure of a system of asynchronously communicating EFSMs can be identified.

The construction of a complete prefix of the labeled event structure is terminated as follows [6]: Before appending a transition of an EFSM $m_i \in M$ to the end of the current configuration, it must be determined whether the last event appended for m_i is a cutoff event. If it is not a cutoff event, then successors of current configuration of m_i can be appended to the labeled event structure. Otherwise, no more events for m_i are appended to the end of the current configuration, unless the wait set of another EFSM $m_j (i \neq j)$ is not empty, and other EFSM $m_j (i \neq j)$ should be considered. If it is found that no more events can be appended to the labeled event structure on any EFSM m_i for the current configuration, the configuration should be terminated. A *cutoff point* is the final global state reached by a terminated configuration. When all the configurations of the labeled event structure terminates, the entire construction process is finished, and the complete prefix of the labeled event structure is obtained. The principle of the termination criterion is that by appending a successor of cutoff events to a prefix of the labeled event structure, a global state would be reached that must be already covered in the prefix. Therefore, all successors of cutoff events can be omitted from the labeled event structure without losing any reachable global state of the system.

2.1.3 Properties of Labeled Event Structures

In a labeled event structure of a system of asynchronously communicating EFSMs, the current global state of the system is not explicitly represented. However, global states of the system are implicitly represented in a distributed manner. Each of the individual EFSMs may be in any "state" along a path of the parallel trees provided all causal predecessors of this "state" have happened before [7]. The global state corresponding to a configuration of the labeled event structure can be computed by reproducing all the events of that configuration within the system of EFSM's starting from the initial state.

Figure 2.3 shows the initial part of the labeled event structure of Example 1 appended with global state information. For the generalized model with at most one queue for each direction between each pair of EFSMs, a global state can be represented by means of a matrix: each element on the diagonal gs_{ii} represents the current state of $EFSM_i$, and each off-diagonal element gs_{ij}, $i \neq j$, represents the contents of the queue from $EFSM_i$ to $EFSM_j$. "-" represents nonexistent queue. "ε" represents empty queue. The last global state in this example is represented by the left bottom matrix, which means that component "t" is in state "ready1", component "m" is in state "ready", component "r" is in state "ready", there is one signal, "mdatreqAK", in the queue "m" to "r", and all the other queues ("t" to "m", "m" to "t", and "r" to "m") are empty.

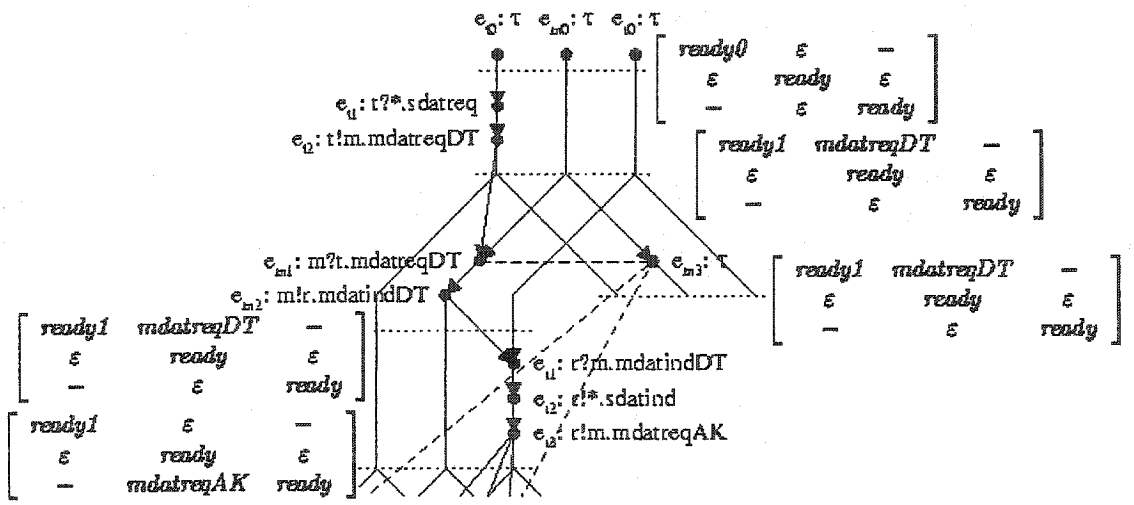


Figure 2.3 Initial Part of the Labeled Event Structure for Example 1 with Global State Information

The algorithm given in [6] generates only a complete prefix of the labeled event structure of a system of asynchronously communicating EFSMs. The labeled event structure is cut off when the system behavior repeats. The generated prefix may be expanded by appending the sub-structure of the existing labeled event structure starting with the corresponding global states to the cutoff points.

A labeled event structure does not hide the difference between non-determinism due to choice of events inside an individual state machine and due to choice of events

from different state machines, as interleaving models do. The choice of events inside an individual state machine is represented by the branching point of the labeled event structure. The non-determinism due to the choice of events from different state machines, which is called concurrency, is settled by the possibility of arbitrary order of events occurring at different state machines [7].

Non-observable transitions τ have no use to build up the tests. They are all omitted from the labeled event structure after the construction.

2.2 Test Derivation Methods

Basically, white box testing can be categorized as two classes: control flow and data flow oriented testing. In general, determining whether an implementation of a distributed system establishes the desired flow of control and data expressed in its specification can be referred to as testing the control and data flow aspects of the implementation. Control flow oriented testing focuses on the analysis of the control flow of the implementation, while data flow oriented testing focuses on the analysis of the data flow of the implementation. The tests constructed by the data flow oriented test selection criteria are complementary to those constructed by control flow oriented test selection criteria [31]. The white box test selection criteria have been adapted to black-box (specification-base) testing for systems specified in Estelle [24], LOTOS [28], and SDL [25]. Since the proposed system aims to produce tests from specifications of concurrent systems for both control flow oriented testing and data flow oriented testing, the commonly used control and data flow oriented testing methods are discussed in the following sections.

The commonly used control-flow and data-flow oriented test generation methods are based on directed graphs (digraphs) representing the source code or the specification of the program under test. As far as applicable, we transfer these methods to graphs representing labeled event structures.

A *path* $(n_1, n_2, \ldots, n_m)$ in a directed graph G is a sequence of nodes in G, such that for all i, $1 \le i \le m\text{-}1$, $m \ge 2$, $(n_i, n_{i+1}) \in E$. A *sub-path* of a path $(n_1, \ldots, n_m)$ is a path $(i_1, \ldots, i_k)$ if there exists a ζ, $0 \le \zeta \le m\text{-}k$, such that for all j, $1 \le j \le k$, $i_j = n_{j+\zeta}$. A *loop-free path* is a

path in which all nodes are distinct. A *complete path* in a single entry, single exit directed graph G is a path whose first node is the entry node and whose last node is the exit node. A complete path is *executable* or *feasible* if input data which causes its execution exists, and un-executable (infeasible) otherwise. A path is executable if it is a sub-path of an executable complete path. Whether a path is executable depends on the semantics of the system itself, not just on the underlying graph structure.

Let p be a complete path in a directed graph G. We say that a node i is *covered* by p if p is a complete path $(n_1, ..., n_m)$ such that $i = n_j$ for some j, $1 \leq j \leq m$. Similarly, an edge (i_1, i_2) is covered by p if p is a complete path $(n_1, ..., n_m)$ such that $i_1 = n_j$ and $i_2 = n_{j+1}$ for some j, $1 \leq j \leq m$-1. A path $(i_1, ..., i_k)$ is covered by a complete path $p = (n_1, ..., n_m)$ if path $(i_1, ..., i_k)$ is sub-path of path $(n_1, ..., n_m)$. A node, edge, or path is covered by a set Π of complete paths of G if the node, edge, or path, respectively is covered by a complete path in Π.

In case of the labeled event structures, we have the problem that it is not sufficient to consider the coverage of test units by paths (i.e. sequences of nodes), but due to concurrency we have to take into account parallel paths (i.e. configurations) in different processes whose events are only partially ordered.

Consider as an example, Figure 2.4 depicting a directed graph $G = (V, E)$ representing a labeled event structure of Example 2, which comes from Example 1 enhanced by data computations. V denotes the set of vertices or nodes; E denotes here the set of directed edges of the graph. For ease of presentation of the following material, the directed graph G in Figure 2.4 has been derived from the graph of a labeled event structure by incorporating a selection of intermediate global states and leaving out the dashed undirected arcs indicating the conflict relation. We also leave aside arrows indicating causal relations between events in different processes. In $G = (V, E)$, the nodes whose identifiers start with "g" indicate global states (e.g., global state node "$g0$"), others indicate labeled events (e.g., event node "$t1$"). In the following discussion, we will use the name of the event as the name of the node as well. For example, we call the node

indicating event "t1", which is labeled with the action "*t?tu.sdatreq(constant, data)*", as node *t*1.

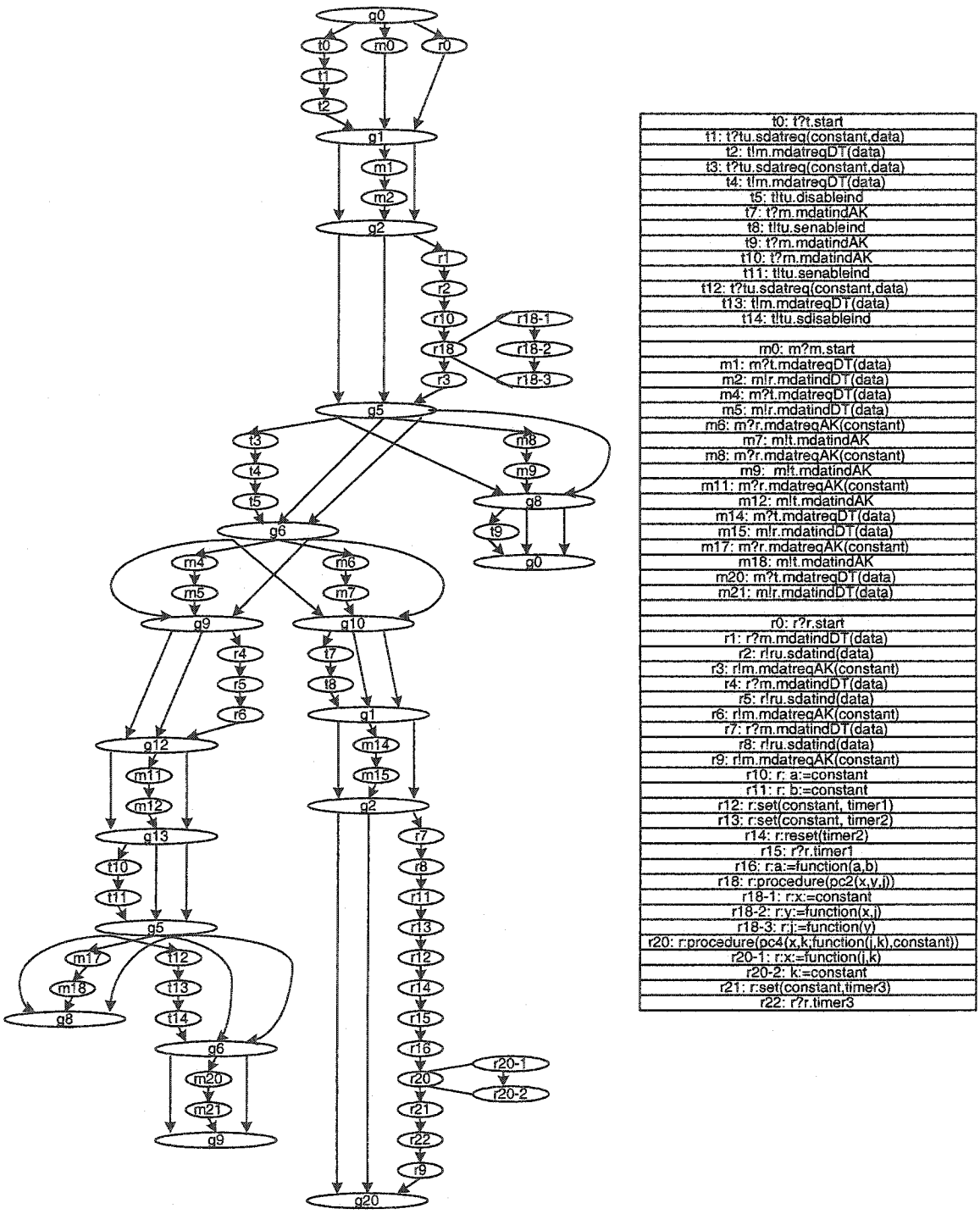


Figure 2.4 Graph G representing a Labeled Event Structure, Example 2

As shown in Figure 2.4, there is no condition statement node and no loop in a graph representing a labeled event structure. The arrows indicate the order of events. Since there are three processes in the system, there are three parallel arcs linking event nodes between two causal global state nodes. We call the global state node indicating the initial states for every process as starting node (e.g., "$g0$" in graph G), and the last global state node of each branch as cutoff node. Every graph representing a labeled event structure can have several branches, therefore, they can have several cutoff nodes.

Let p be a maximal configuration of a given labeled event structure. We say that an event i is *covered* by p if $i \in p$. Similarly, a path $(i_1, ..., i_k)$ is covered by a maximal configuration p if $i_j \in p$ for all $1 \leq j \leq k$. A node or path is covered by a set Π of maximal configurations if the node or path is covered by a maximal configuration in Π.

2.2.1 Control Flow Oriented Testing

Each control flow oriented testing method aims at generating a set of test cases that covers (i.e. causes execution of) all occurrences of a control flow oriented test unit (e.g. node, edge, branch, outcome of a decision, etc.). There are various control flow oriented testing methods: all-nodes [4, 10, 18], all-edges [4, 10, 18], all-paths [4, 10, 18], decision coverage (branch coverage), condition coverage, decision/condition coverage and multiple condition coverage. Let Π be a set of complete paths of a directed graph G.

- All-nodes

Π satisfies the all-nodes coverage criterion in G if each node in G is covered by Π at least once. This is the easiest and weakest of all the testing methods.

- All-edges

Π satisfies the all-edges coverage criterion in G if each edge in G is covered by Π at least once.

• All-paths

Π satisfies the all-paths coverage criterion in G if each complete path in G is in Π. This is the strongest criterion among the testing methods, which is not generally practical. But in a graph representing a labeled event structure, since there are no conditions and no loops, it is possible to achieve all-paths coverage.

• Decision Coverage

A decision is a boolean expression [18]:

<decision> := <component> AND <component>
| <component> OR <component>
| NOT <component>

A component of a decision can be a decision or a condition, which is defined as follows:

<condition> := <var1> = <var2>
| <var1> > <var2>
| <var1> < <var2>
| <var1> >= <var2>
| <var1> <= <var2>
| <var1> != <var2>
| etc.

where var1, var2 may be optionally replaced by function names or constants.

Π satisfies the decision coverage criterion [18] (or branch coverage criterion) in G if each outcome of each decision (i.e. branch) in G is covered by Π at least once.

However, if there is more than one condition in a decision, decision coverage may not expose the errors that could be present in a particular condition. For example, for testing the decision (A or B), setting A to true and B to false in one test case, and A to false and B to false in another would satisfy the decision coverage criterion, but it could

mask a possible error in condition *B*. Thus, a shortcoming of the decision coverage criterion is that it does not guarantee coverage of each condition within each decision.

- Condition Coverage

Π satisfies the condition coverage criterion [18] in *G* if every outcome of each condition in each decision in *G* is covered by Π at least once. Although the condition coverage criterion states that every outcome of each condition in each decision will be executed, it does not however guarantee that all outcomes of every decision will be taken and hence it does not guarantee the coverage of all statements or all branches in a graph. For example, for testing the decision (*A* or *B*), setting *A* to true and *B* to false in one test case, and *A* to false and *B* to true in another would satisfy the condition coverage criterion, but it would not satisfy the decision coverage criterion, since it would fail to test the false outcome of the decision.

- Decision/Condition Coverage

Π satisfies the decision/condition coverage criterion [18] in *G*, if each outcome of each condition and each outcome of each decision in *G* is covered by Π at least once. To compensate for the deficiencies of condition coverage and decision coverage, decision/condition coverage ensures that each outcome of each condition and each outcome of each decision will be executed. However, it is still possible not to uncover the error of having "and" instead of "or" in a decision. For example, for testing the decision (*A* and *B*), setting *A* and *B* both to true in one test case, and *A* and *B* both to false in another would satisfy the decision/condition coverage criterion, but it would be possible to replace the "and" for an "or" without affecting the result of the tests.

- Multiple Condition Coverage

Π satisfies the multiple condition coverage criterion [18] in *G*, if each possible combination of outcomes of conditions in each decision in *G* is covered by Π at least once. Multiple condition coverage eliminates the deficiencies of the previous criteria, since test cases must be generated to exercise all possible combinations of all condition outcomes in each decision.

It is widely accepted that all-nodes coverage is rather weak, and branch coverage criterion is a minimal standard of achievement in white box testing [31]. Other criteria (multiple condition, decision/condition, condition) are difficult to achieve in software of more than moderate complexity [31].

Only few of the test methods apply to labeled event structures. All-nodes, all-edges, and all-path (or all-maximal-configuration) coverage all boil down to the same due to the tree-like structure of labeled event structures. Decision, condition, decision/ condition, and multiple condition coverage are not applicable because there are no decisions and conditions in the model.

2.2.2 Data Flow Oriented Testing

Data flow oriented testing is based on data flow analysis. Data flow analysis tracks variables as they are defined and modified until they are used to compute values for other variables or to produce output values. Data flow oriented testing is based on the analysis of the data flow characteristics of the software being tested. Data flow oriented test generation criteria are constructed so that critical associations between the definition of a variable and its uses are examined during software testing. The reason for 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 the result of propagation through computations, an error may show up at the system's output. Just as one would not feel confident about the correctness of a portion of a software which has never been executed, it is believed that if the result of some computation has never been used, one has no reason to conclude that the correct computation has been performed.

2.2.2.1 Data Flow Related Concepts

Data flow analysis was originally used for compiler optimization [33]. It aims to trace the program variables as they are initialized and modified during the program execution. Data flow analysis classifies each variable occurrence in a statement associated with a node or in a Boolean expression associated with an edge in a graph representing the software as a definition or a use [10, 22]. A *definition* (or def) of a variable x at node n

(denoted by d^x_n) is an occurrence of x by which x receives a value (e.g., an occurrence of x on the left hand side of assignment statement, for instance, the definition of "a" at node "$r10$" in Figure 2.4). A *use* of a variable x is an occurrence of x by which the value of x is referenced. Uses of variables are further classified as computational uses (c-uses) and predicate uses (p-uses) [10, 22]. A *c-use* of a variable x at node n (denoted by c^x_n) is a use of x which directly affects the computation being performed (e.g., an occurrence of x on the right hand side of an assignment statement, for instance, the use of "y" at node "$r18$-3" in Figure 2.4) or allows one to see the result of some earlier definition (e.g., an occurrence of x in the list of variables of an output statement, for instance, the use of data at node "$t2$" in Figure 2.4). A *p-use* of a variable x on edge (n, m) (denoted by $p^x_{(n, m)}$) is a use of x which directly affects the control flow (i.e., an occurrence of x in the Boolean expression associated with an outgoing edge of a node n).

2.2.2.2 Classification of Variable Occurrences

In a graph representing a labeled event structure, a sequence of definitions and/or c-uses is associated with each node.

(a) an input event $e(X_1, \ldots, X_n)$ in process P, where $X_1, \ldots, X_n$ are local to P, contains defs of the variables $P.X_1, \ldots, P.X_n$. If the input signal is a timeout signal, t, then it contains a c-use of the timer-identifier, $P.t$.

(b) an output event $e(X_1, \ldots, X_n)$ in process P, where $X_1, \ldots, X_n$ are local to P, contains c-uses of the variables $P.X_1, \ldots, P.X_n$.

(c) an assignment statement Y := expression in process P contains c-uses of the variables occurring in the expression followed by a def of Y.

(d) a set statement set(expression, t) in process P contains c-uses of the variables occurring in the expression followed by a def of the timer-identifier, $P.t$.

(e) a reset statement reset(t) in process P contains a c-use of the timer-identifier, $P.t$.

(f) a procedure call statement $\pi(X_1, \ldots, X_m; e_{m+1}, \ldots, e_n)$ in process P contains, for each variable X_i, $(1 \leq i \leq m)$,

- c-uses of Z_j, $1 \leq j \leq r$, followed by def of $P.X_i$ which is defined in the abstract definition of π by $P.X_i$ ":=" function($Z_1, \ldots, Z_r$), Z_j is $P.X_s$ or a variable in the expression e_t, $1 \leq s \leq m$, $m+1 \leq t \leq n$; or

- def of $P.X_i$ which is defined in the abstract definition of π by $P.X_i$:= constant.

Note that:

1) expression e is function($v_1, \ldots, v_n$) or constant, where $v_1, \ldots, v_n$ are variables.

2) A given LES refers to abstract functions, expressions and procedure calls because the corresponding system specification has the actual functions, expressions and procedure calls.

The following table shows the definitions and c-uses in the graph G for Example 2 in Figure 2.4.

Table 2.1 defs and c-uses for Figure 2.4

Event	Action	Defs	C-uses
t0	t?t.start		
t1	t?tu.sdatreq(constant,data)	t.data	
t2	t!m.mdatreqDT(data)		t.data
t3	t?tu.sdatreq(constant,data)	t.data	
t4	t!m.mdatreqDT(data)		t.data
t5	t!tu.disableind		
t6	t?t.nil		
t7	t?m.mdatindAK		
t8	t!tu.senableind		
t9	t?m.mdatindAK		
t10	t?m.mdatindAK		
t11	t!tu.senableind		
t12	t?tu.sdatreq(contant, data)	t.data	
t13	t!m.mdatreqDT(data)		t.data
t14	t!tu.sdisableind		
m0	m?m.start		

m1		m?t.mdatreqDT(data)	m.data	
m2		m!r.mdatindDT(data)		m.data
m4		m?t.mdatreqDT(data)	m.data	
m5		m!r.mdatindDT(data)		m.data
m6		m?r.mdatreqAK(constant)		
m7		m!t.mdatindAK		
m8		m?r.mdatreqAK(constant)		
m9		m!t.mdatindAK		
m11		m?r.mdatreqAK(constant)		
m12		m!t.mdatindAK		
m14		m?t.mdatreqDT(data)	m.data	
m15		m!r.mdatindDT(data)		m.data
m17		m?r.mdatreqAK(constant)		
m18		m!t.mdatindAK		
m19		m?m.nil		
m20		m?t.mdatreqDT(data)	m.data	
m21		m!r.mdatindDT(data)		m.data
r0		r?r.start		
r1		r?m.mdatindDT(data)	r.data	
r2		r!ru.sdatind(data)		r.data
r10		r:a:=constant	r.a	
r18		r:procedure(pc2(x,y,j;a))		
	r18-1	{x:=function(a);	r.x	r.a
	r18-2	y:=function(x);	r.y	r.x
	r18-3	j:=constant;}	r.j	
r3		r!m.mdatreqAK(constant)		
r4		r?m.mdatindDT(data)	r.data	
r5		r!ru.sdatind(data)		r.data
r6		r!m.mdatreqAK(constant)		
r7		r?m.mdatindDT(data)	r.data	
r8		r!ru.sdatind(data)		r.data
r11		r:b:=constant	r.b	
r12		r:set(constant,timer1)	r.timer1	
r13		r:set(constant,timer2)	r.timer2	
r14		r:reset(timer2)		r.timer2
r15		r?r.timer1		r.timer1
r16		r:a:=function(a,b)	r.a	r.a, r.b
r20		r:procedure(pc4(x,j;y))		
	r20-1	{x:=function(j,y);	r.x	r.j, r.y
	r20-2	j:=constant;}	r.j	
r21		r:set(constant,timer3)	r.timer3	
r22		r?r.timer3		r.timer3
r9		r!m.mdatreqAK(constant)		

A path $(n_1, n_2, \ldots, n_{m-1}, n_m)$ is a *def-clear path* with respect to a variable x from node n_1 to node n_m or from node n_1 to edge (n_{m-1}, n_m) if there are no definitions of x at nodes n_2 to n_{m-1}.

Data flow information in a graph G is organized by associating with each node i the sets c-use(i) = {variables which have c-uses in node i} and def(i) = {variables which have definitions in node i}, and associating with each edge (i, j) the set p-use(i, j) = {variables which have p-uses on edge (i, j)}. We also define sets of nodes dcu(x, i) = {node j such that $x \in c$-use(j) and there is a def-clear path which respect to x from i to j} and dpu(x, i) = {edge (j, k) such that $x \in p$-use(j, k) and there is a def-clear path with respect to x from i to (j, k)}.

A def-use association is also called *du-pair*. Du-pairs can be categorized into two classes: def-c-use association and def-p-use association. A *def-c-use* association is a triple (i, j, x) where i is a node containing a definition of x and $j \in$ dcu(x, i). A *def-p-use* association is a triple $(i, (j, k), x)$ where i is a node containing a definition of x and $(j, k) \in$ dpu(x, i). A *def-use association* is a def-c-use association or def-p-use association. We say that a def-use association is feasible (executable) if a def-clear path related to the association is a sub-path of some executable complete path; otherwise, it is infeasible (unexecutable).

A path $(n_i, \ldots, n_j, n_k)$ is a *du-path* with respect to a variable x if n_i has a definition of x and either

a) n_k has a c-use of x and $(n_i, \ldots, n_j, n_k)$ is a def-clear path with respect to x, or

b) (n_j, n_k) has a p-use of x and $(n_i, \ldots, n_j, n_k)$ is a def-clear loop-free path with respect to x.

On the other hand, in a graph representing a labeled event structure, there is no Boolean expression on edges, so there are no p-uses in the graph. Therefore, in this thesis, only definitions and c-uses are considered in a graph representing a labeled event structure where there are no conditions and thus all paths are executable and all def-use

associations are executable. For the example graph in Figure 2.4, the def-c-use associations for each def of each variable as well as the corresponding def-clear paths are given in Table 2.2.

Table 2.2 du-pairs for Figure 2.4

Variable name	def at	c-use at	def-clear path
t.data	t1	t2	t1->t2
t.data	t3	t4	t3->t4
t.data	t12	t13	t12->t13
m.data	m1	m2	m1->m2
m.data	m4	m5	m4->m5
m.data	m14	m15	m14->m15
m.data	m20	m21	m20->m21
r.data	r1	r2	r1->r2
r.data	r4	r5	r4->r5
r.data	r7	r8	r7->r8
r.a	r10	r18-1	r10->r18
r.a	r10	r16	r10->r18-1->r18-2->r18-3->r3->r7->r8->r11->r13->r12->r14->r15->r16
r.x	r18-1	r18-2	r18-1->r18-2
r.y	r18-2	r20-1	r18-2->r18-3->r3->r7->r8->r11->r13->r12->r14->r15->r16->r20-1
r.j	r18-3	r20-1	r18-3->r3->r7->r8->r11->r13->r12->r14->r15->r16->r20-1
r.b	r11	r16	r11->r13->r12->r14->r15->r16
r.timer1	r12	r15	r12->r14->r15
r.timer2	r13	r14	r13->r12->r14
r.timer3	r21	r22	r21->r22

2.2.2.3 Data Flow Oriented Test Selection Criteria

Data flow oriented testing has been motivated to bridge the gap between all-paths and all-edges coverage criteria [10, 12]. A family of data flow oriented test selection criteria is defined in [22]. The criteria attempt to cover various combinations between defs and uses of variables. Roughly speaking, the family of data flow oriented test selection criteria is based on the requirement that the test cases execute def-clear paths from each node containing a definition of a variable to specified nodes containing *c*-uses and edges containing *p*-uses of that variable. For each variable definition, it is required that all/some def-clear paths with respect to that variable from the node containing the definition to

all/some of the uses/c-uses/p-uses reachable by some such paths be executed. These criteria are defined precisely in Table 2.3.

Table 2.3 Some Data Flow Oriented Test Selection Criteria

Criterion	Association Required
All-defs	Some (i, j, x) such that $j \in \text{dcu}(x, i)$ or some $(i, (j, k), x)$ such that $(j, k) \in \text{dpu}(x, i)$.
All-c-uses	All (i, j, x) such that $j \in \text{dcu}(x, i)$.
All-p-uses	All $(i, (j, k), x)$ such that $(j, k) \in \text{dpu}(x, i)$.
All-p-uses/some-c-uses	All $(i, (j, k), x)$ such that $(j, k) \in \text{dpu}(x, i)$. In addition, if $\text{dpu}(x, i) = \phi$ then some (i, j, x) such that $(j, k) \in \text{dcu}(x, i)$. Note that since i has a definition of x, $\text{dpu}(x, i) = \phi \Rightarrow \text{dcu}(x, i) \neq \phi$.
All-c-uses/some-p-uses	All (i, j, x) such that $j \in \text{dcu}(x, i)$. In addition, if $\text{dcu}(x, i) = \phi$ then some $(i, (j, k), x)$ such that $j \in \text{dpu}(x, i)$. Note that since i has a definition of x, $\text{dcu}(x, i) = \phi \Rightarrow \text{dpu}(x, i) \neq \phi$.
All-uses	All (i, j, x) such that $j \in \text{dcu}(x, i)$ and all $(i, (j, k), x)$ such that $(j, k) \in \text{dpu}(x, i)$.
All-du-paths	All du-paths from i to j with respect to x for each $j \in \text{dcu}(x, i)$ and all du-paths from i to (j, k) with respect to x from each $(j, k) \in \text{dpu}(x, i)$.

If variable x has a definition in node i, the all-defs coverage criterion requires the test cases to exercise some def-clear path wrt x from node i to some node or edge at which the value assigned to x in node i is used. The all-uses coverage criterion requires the test cases to exercise at least one such path to each such node and to each such edge. The all-du-paths coverage criterion requires that all of the du-paths from node i to each such node and each such edge be exercised. The coverage criteria all-p-uses, all-c-uses, all-p-uses/some-c-uses, and all-c-uses/some-p-uses place emphasis on either c-uses or p-uses. Note that any program has only finite set of def-use association, so none of the data flow oriented test selection methods requires infinitely many test units. Figure 2.5 illustrates the subsumption relationships between control and data flow oriented test selection criteria. A criterion A *subsumes* another criterion B, denoted $A \Rightarrow B$, if a set of complete paths Π satisfying A also satisfies B. As we can see from Figure 2.5, every data flow oriented test selection criterion is stronger than all-edges coverage criterion. To find

out more about the advantages of data flow oriented testing over control flow oriented testing, an interested reader may refer to reference [4, 9, 10] for detail.

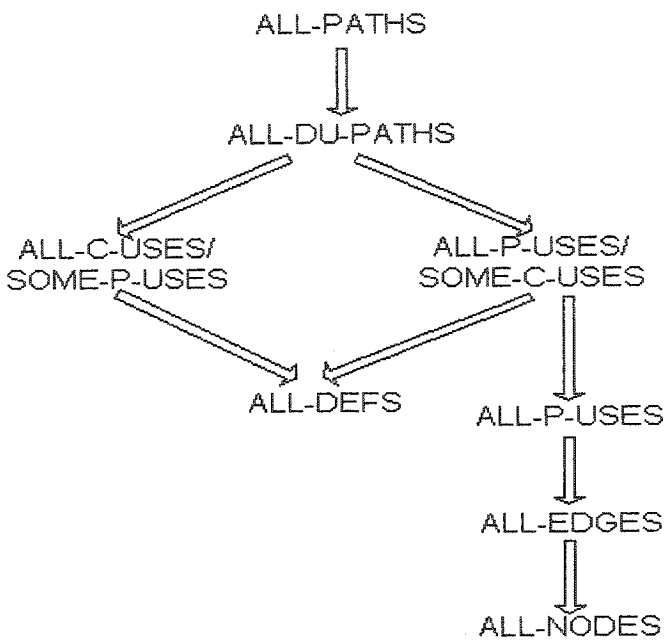


Figure 2.5 Relationships Among Test Selection Criteria

Chapter 3 Test Derivation from Labeled Event Structures

3.1 Data Structure for Labeled Event Structure

To construct a set of configurations from a given labeled event structure, a set of internal data structures is used to store and process the labeled event structure. The following is the list of components of the data structure describing a labeled event structure:

I) action set (A) stores all the actions that can occur in the system of asynchronously communicating EFSMs.

Action indicates the action taken by an EFSM. It contains the following fields:

- the name of the EFSM, called local component (loc), at which the action occurs.
- the mode of the action (mode). In this thesis, the prototype tool considers the following modes of actions:
    - input, which is marked by "?";
    - output, which is marked by "!";
    - assignment, which is marked by "=";
    - set timer, which is marked by "s";
    - reset timer, which is marked by "r";
    - timeout, which is also marked by "?". It can be distinguished from an input action by that the local component name and remote component name of a timeout action are always the same;

- procedure call, which is marked by "p". In our approach, parser breaks each procedure call action into several assignments according to the abstract definition of the called procedure. For example, procedure "procedure(pc4(x,j;y)){x:=function(j,y);j:=constant;}" is broken into two assignments: "x:=function(j,y)" and "j:=constant".

- the name of the remote component (rem). This field is only used by input, output and timeout actions. For an input action "loc?rem.msg", the remote component is the component of the concurrent system that sent out the message "msg"; for an output action "loc!rem.msg", the remote component is the component that receives the message "msg"; for a timeout action, the remote component is always the same as the local component, where the timer identifier is used.

- the name of the message (msg). This field is only used by input, output and timeout actions. For input and output actions, this field stores the signal name; for timeout actions, this field stores the name of the timer identifier.

- the parameter list is a set of names (paras). In input and output actions, this field stores a list of names of the signal parameters. If the signal has no parameters, the parameter list is empty. In assignment actions, the first element of the parameter list is always the name of the variable, which is assigned a value in the assignment statement, followed by a list of names of variables which occur in the expression of the assignment statement. In an assignment statement whose expression is a constant, the parameter list contains only one element, which is the variable on the left side of the assignment symbol. In set and reset actions, the parameter list stores the timer identifier only. In timeout actions, the parameter list is empty, and the timer identifier is stored in the message (msg) field. Table 3.1 shows some example for actions with their corresponding content of the parameter list fields:

Table 3.1 Actions and Parameter List:

Action	Content of parameter List
m?t.mdatreqDT(data)	m.data
t!tu.disableind	∅
r: a:=constant	r.a

r: x:=function(j,y)		r.x, r.j, r.y
r: set(constant, timer1)		r.timer1
r: reset(timer2)		r.timer2
r? r.timer1		∅
procedure(pc4(x,j;y)) {x:=function(j,y); j:=constant;}	x:=function(j,y);	r.x, r.j, r.y
	j:=constant;	r.j

- list of definitions is a set of variables (defs), which are defined in the current action. The details for classification of definitions are in 2.2.2.2.

- list of *c*-uses is a set of variables (uses), which are used in the current action. The details for classification of c-uses are in 2.2.2.2.

Both definitions and *c*-uses belong to the data dependency. The data Dependency is used to store the information on the classification of variable occurrences. It has two fields:

- the name of the variable (var).

- the classification of the variable occurrence (mode): "d" indicates the definitions and "c" indicates the *c*-uses. *p*-uses do not exist in labeled event structures.

For example, action "a:=function(a,b)" has the following Data Dependency data structure:

Table 3.2 Data Structure for Data Dependency

The name of the variable	The mode of the variable occurrence
a	c
b	c
a	d

II) event set (E) stores all the events in the labeled event structure.

Event is the prime element of a labeled event structure. It contains the following fields:

- the identification of the event (id), which is also called the name of the event. The name of the event is composed of the name of the component followed by a number indicating the order of the event occurring in the labeled event structure. Therefore, each event has a unique name in a labeled event structure.

- the action of the event (act). This field is a pointer, pointing to an action structure described above.

- conflict events (conflicts) stores a set of events, which are marked as conflicting with the current event at some branching point of the labeled event structure. Although the conflict relationship is inherited, we only store the direct conflict events in the field.

- predecessor events (predecessors) is a set of events, which are the immediate causal predecessors of the current event. Normally, at least one predecessor occurs at the same component as the current event (the exceptions are: start event of each component (e.g., t?t.start) has no same-component predecessor; timeout or timer reset event may have two same-component predecessors: one is the immediate causal predecessor event, the other is the corresponding set event). Other causal predecessors which occur at different components are output events whose sent out messages are received by the current event.

- successor events (successors) is a set of events, which are the immediate causal successors of the current event. Normally, at least one successor occurs at the same component as the current event (the exceptions are: cutoff events have no same-component successor; set events may have two same-component successors). Events before branching points may have more than one immediate same-component successor. Other causal successors which occur at different components are input events which receive messages from the current event.

- def-used set (usedefs) stores a set of variables, which are defined in the current event and have been used in some du-pair in the labeled event structure.

Each event in a labeled event structure has a unique id. Different events can point to the same action, in the case that a transition of a system of asynchronously communicating EFSMs occurs in a labeled event structure more than once. The causal relationship between two events is set as a pair. For example, if event "a" has event "b" as a causal successor event, "b" must have "a" as a causal predecessor event.

III) component set (C) stores the names of all the components and test components in the system of asynchronously communicating EFSMs.

IV) message set (Mg) stores all the messages that are used in the system of asynchronously communicating EFSMs.

V) root (Root) is a set of initial events, each of which is for one component of the system of asynchronously communicating EFSMs. In the graph representing a labeled event structure in Figure 2.4, the root is "t0, m0, r0". The test components (i.e., environment) are not included.

VI) cutoff points (cutoffps) set stores all the cutoff points of the constructed labeled event structure. Each cutoff point contains a set of cutoff events, each component of the system of asynchronously communicating EFSMs has one cutoff event in a cutoff point. Every maximal configuration in the labeled event structure has a corresponding cutoff point. For example, in the graph representing a labeled event structure in Figure 2.4, there are four configurations, therefore, there are four cutoff points existing for the labeled event structure: "t11, m18, r6", "t14, m21, r6", "t8, m15, r9" and "t9, m9, r3". Each cutoff point has a corresponding point in the labeled event structure, which has the same global states as the cutoff point (e.g., the corresponding point of "t11, m18, r6" is "t6, m9, r3").

VII) du-pair set (dups) stores all the du-pairs in the labeled event structure.

The du-pair is used to trace the definition and use of a variable in a labeled event structure. It has four fields:

- the name of the variable x (var).

- def event (defev) is the event where the variable x is defined. Variables can be defined and re-defined more than once in a labeled event structure. Theoretically speaking, each definition should have at least one corresponding du-pair in a well-formed graph, which means each defined variable should be used at least once in the system.

- c-use event (usev) is the event where variable x is c-used, and there is a def-clear path from the def event to the c-use event with respect to x.

- def-clear path (dcp) is a sequence of events, recording the def-clear path with respect to x. The first event is the def event, and the last event is the c-use event. For local variables, all the events in a def-clear path occur at the same component. The events in a def-clear path must belong to a configuration of the labeled event structure.

For example, in graph G in Figure 2.4, one du-pair in component "t" is that variable "t.data" is defined at event "t1" and c-used at event "t2". It could be represented in du-pair data structure as:

Table 3.3 Data Structure for du-pair

variable	def event	c-use event	def-clear path
t.data	t1	t2	t1, t2

VIII) control flow configuration set (cconfs) stores a set of configurations for the control flow oriented testing of the given labeled event structure.

IX) data flow configuration set (dconfs) stores a set of configurations for the data flow oriented testing of the given labeled event structure.

As defined in Section 2.1.1, a *configuration* is a causally closed, conflict-free subset of events of a labeled event structure. Hence, a configuration is a set of events partially ordered by the causality relation. For each test case, a configuration data structure is used to generate and store the partially ordered set of events that the test

should follow. This data structure works for both control flow and data flow oriented testing. It has four fields:

- the event set (E) contains all the events of the configuration. This set of events is a subset of the event set of the labeled event structure.

- the component set (C) contains the names of both components and test components (i.e., environment) of the system under test. The content of this field is the same as the content of the component set of the labeled event structure.

- cutoff point (cutoffp) is a set of events. Each component of the system under test has one cutoff event for one configuration. The cutoff point for each configuration is unique.

- linearization (conf) is a linearization of the events (local to each component) in the event set of the configuration that will be used to generate the output MSC. The linearization (conf field) has exactly the same events as the event set of the configuration. The order of the events per component in conf will be used to generate the output MSC.

This model does not hide the concurrency attribute of a labeled event structure by the order of events. In other words, the order in the linearization (conf field) of a configuration

i) is to be strictly followed for events occurring in each individual component;

ii) is not to be strictly followed for events occurring in different components since they can occur in any order if they have no causal relationship.

For example, in the graph representing the labeled event structure of Figure 2.4, the Configuration data structure for the left most configuration for control flow oriented testing is as follows:

Table 3.4 Data Structure for Configuration

Field	Content
event set	t11, m18, r6, t10, m17, r5, t5, m12, r4, t4, m11, m5, r3, t3,

	m4, r18-1, r18-2, r18-3, t2, m2, r10, t1, m1, r2, t0, m0, r1, r0
component set	t, m, r, tu, ru
cutoff point	t11, m18, r6
conf	t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, t5, m4, m5, r4, r5, r6, m11, m12, t10, t11, m17, m18

In this configuration, event "m2" and "t3" have no causal relationship, and the order of occurrence of these two events in the given linearization of the configuration can be arbitrarily established when forming the output MSC.

3.2 Configuration Selection for a Test Unit

A *test unit* is a node, an edge or a def-c-use association, etc. A file containing all the test units based on a given test selection criterion is called a *test unit file*. The types of test units produced will depend on the chosen test selection criterion. These are summarized in Table 3.5.

Table 3.5 Test Units for Some Test Selection Criteria

Testing Selection Criteria	Test Unit
All-nodes	Node n_i
All-edges	Edge (n_i, n_j)
All-uses	Def-c-use association

In our approach, we use the all-nodes test selection criterion for control flow oriented testing and the all-uses test selection criterion for data flow oriented testing.

There are usually many complete paths to cover a test unit within a software. Sometimes, the number of complete paths covering a test unit may even be infinite and the length of some of these paths may be unbounded. In a labeled event structure, both the number and the length of maximal configurations are finite.

Figure 3.1 is the labeled event structure of Example 2. Apart from those maximal configurations with a maximal element that is labeled with the unobservable action τ (e_{m3}, e_{m10}, e_{m13}, e_{m16}, or e_{t15} respectively), which can be omitted without loss of information, there exist four maximal configurations whose linearizations are given below:

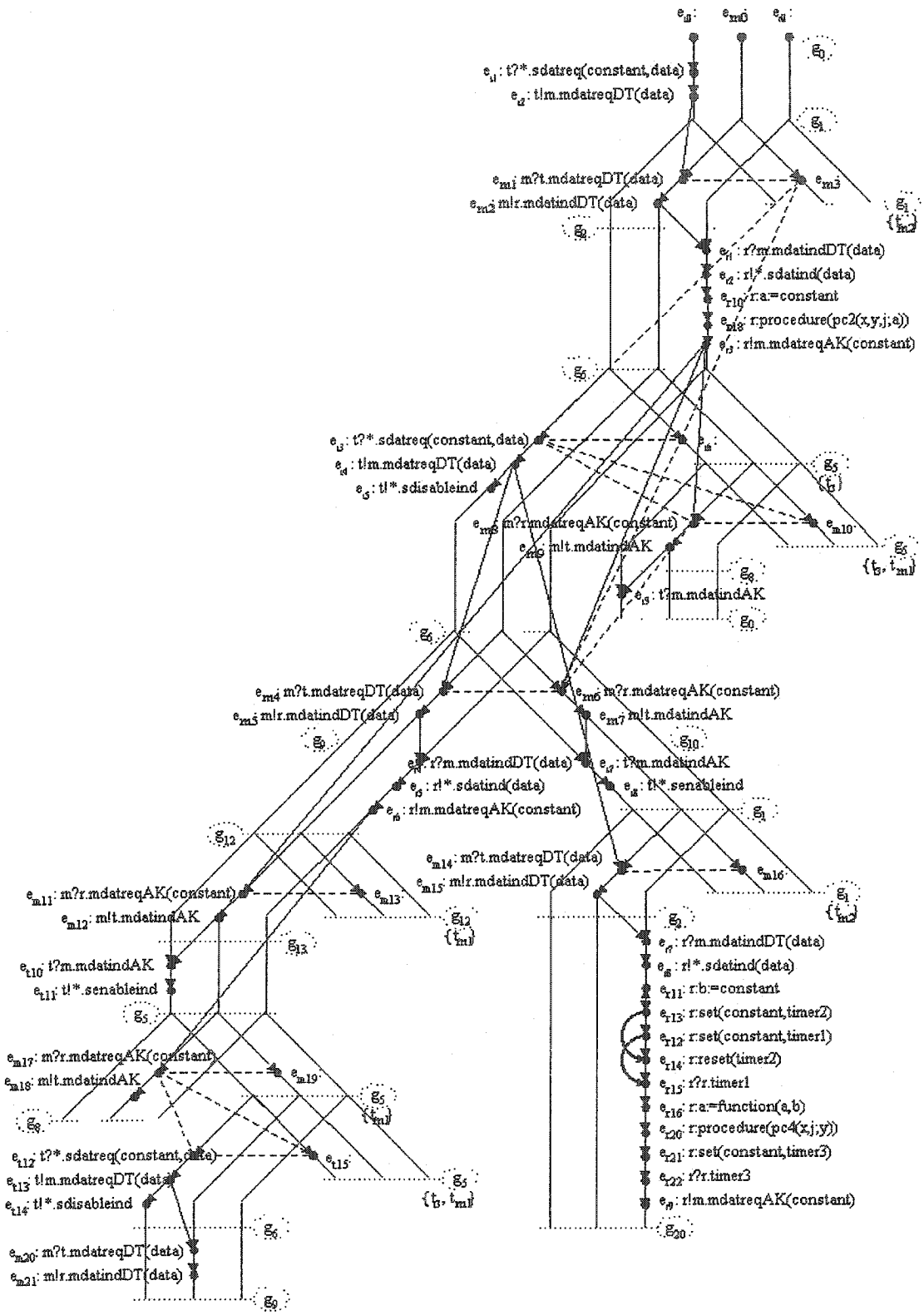


Figure 3.1 Detailed Labeled Event Structure of Example 2

(1) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, t5, m4, m5, r4, r5, r6, m11, m12, t10, t11, m17, m18;

(2) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, t5, m4, m5, r4, r5, r6, m11, m12, t10, t11, t12, t13, t14, m19, m20, m21;

(3) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, t5, m6, m7, t7, t8, m14, m15, r7, r8, r11, r13, r12, r14, r15, r16, r20-1, r20-2, r21, r22, r9;

(4) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t6, m8, m9, t9;

As shown in Figure 3.1, node "r16" is covered by configuration 3; node "m5" is covered by both configuration 1 and configuration 2; a def-c-use association in component "r" on variable "r.data", which is defined at node "r4" and c-used at node "r5", through the def-clear configuration "r4->r5", is covered by both configuration 1 and configuration 2.

In our approach, we use the all-nodes test selection criterion for control flow oriented testing. As we discussed in Section 2.2.1, the test units of the all-nodes test selection criterion are the nodes in the complete prefix of the labeled event structure, which are the events occurring in the system, up to recurrence. The all-nodes test selection criterion fulfills the requirement that all the events of the complete prefix of a labeled event structure are to be tested.

All-uses test selection criterion is used for data flow oriented testing in this thesis. The set of configurations not only covers all the def-use associations in the labeled event structure, but also covers all the definitions which are not being used in the constructed labeled event structure. The way to cover those un-used definitions is to append a sub-structure to the labeled event structure in order that new def-use associations are built up and those unused definitions are put into use in the new def-use associations. Then the set of configurations for data flow oriented testing has to cover these new def-use associations. To ensure a def-use association is executable, the events for the definition and the *c*-use, as well as all other events in the def-clear path of the def-use association must occur in a maximal configuration of the labeled event structure.

The approach to construct the set of configurations from a labeled event structure that satisfies the above test selection criteria is discussed in the following sections.

3.3 Maximal Configuration Generation for Control Flow Oriented Testing

In our approach, the control flow oriented test generation from the complete prefix of the labeled event structure of a system of asynchronously communicating EFSMs is carried out in the following steps:

1. Identify all maximal configurations of the prefix;
2. Restrict the events in configurations to only those occurring at the points of control and observation (PCOs) of the test environment;
3. For each maximal configuration, check whether it is included in another maximal configuration. If it is, then eliminate it from the set of maximal configurations.

The algorithm for control flow oriented test generation in terms of data structures defined for a labeled event structure is outlined as follows.

First, the data structures are initialized. One unique configuration will be generated for each cutoff point of the labeled event structure. For each configuration, the component set field contains the same component names as the component set of the labeled event structure, and the cutoff point field is one of the cutoff points of the labeled event structure.

In order to obtain the set of events occurring in a configuration, we need an event queue. All the events in the cutoff point field of the configuration should be added into the event queue and to the event set of the configuration. Loop while the event queue is not empty, dequeue the first event from the queue and put its predecessors (which do not appear yet in the event set of the configuration), into the event queue and to the event set of the configuration. When the loop terminates, the event queue is empty and all the

events occurring in the configuration have been put into the event set of the configuration.

In order to build each configuration, first, all the events in the event set of the configuration are put into an initially empty queue. Loop while the event queue is not empty, dequeue the first event from the queue, check the interaction set of its successors with the event set of the configuration. If all the events in the interaction set are already included in the configuration, then add the event into the configuration, otherwise, enqueue the event into the event queue. After a configuration is built, it is added into the control flow oriented configuration set.

The process of restricting events to only those occurring at the PCOs consists of checking the events in each maximal configuration and omitting the events for which neither the local component nor the remote component is a tester component. The restriction has to be done because only the events occurring at the PCOs can be observed or controlled by the tester.

At the end, in order to get a minimal set of maximal configurations, each configuration is checked whether it is included in another maximal configuration in the obtained configuration set.

The algorithm is described in pseudo-code as follows:

Given a complete prefix $\sigma = \langle E, \preceq, \#, l \rangle$ of the labeled event structure of a system of asynchronously communicating EFSMs m, (the complete prefix is represented in the internal data structure defined in 3.1), construct a set of maximal configurations which satisfies the all-nodes test selection criterion for control flow oriented testing of the given complete prefix of the labeled event structure.

Algorithm

Step 1. Identify all maximal configurations of the complete prefix

cconfs := $\varnothing$;

```
for all cutoffp_i ∈ σ.cutoffps do
    cconf_i := ⟨∅,∅,∅,∅⟩;
    cconf_i.C := σ.C;
    cconf_i.cutoffp := cutoffp_i;
    pred_queue := ∅;
    /// get the complete event set (cconf_i.E) for the configuration
    for all e_j ∈ cutoffp_i do
        cconf_i.E := cconf_i.E ∪ e_j;
        put(pred_queue, e_j);
    endfor;
    while not empty(pred_queue) do
        ev := get(pred_queue);
        for all e_j ∈ ev.predecessors do
            if e_j ∉ cconf_i.E then
                cconf_i.E := cconf_i.E ∪ e_j;
                put(pred_queue, e_j);
            endif;
        endfor;
    endwhile;
    /// generate a maximal configuration from the event set
    backward_conf := ∅;
    e_queue := ∅;
    for all e_j ∈ cconf_i.E do
        put(e_queue, e_j);
    endfor;
    while not empty(e_queue) do
        ev := get(e_queue);
        if ( (ev.successors ∩ cconf_i.E) ⊆ backward_conf) then
            backward_conf := backward_conf ∪ ev;
        else
            put(e_queue, ev);
```

```
            endif
        endwhile
        for all e_j∈ backward_conf (j values from backward_conf.size()-1 to 0) do
            cconf_i.conf := cconf_i.conf ∪ e_j;
        endfor;
        cconfs := cconfs ∪ cconf_i;
endfor;
```

Step 2. Restrict the events in configurations to only those occurring at the PCOs

Events occurring at the PCOs are those events that interact with the environment. The component field of a labeled event structure includes not only the names of the components of the system, but also those of the environment. All the component names (comps) of the system can be known from any root point or any cutoff point of the labeled event structure. So all the other names in $\sigma.C$ are test component names (Tcomps). To restrict the events to only those occurring at the PCOs is to omit the events whose local component and remote component are both not those of environment (also named test components (Tcomps) in our approach).

```
for all cconf_i∈ cconfs do
    for all e_j∈ cconf_i.conf do
        if ((e_j.act.loc∉ Tcomps) and (e_j.act.rem∉ Tcomps)) then
            cconf_i.E := cconf_i.E - e_j;
            cconf_i.conf := cconf_i.conf - e_j;
        endif;
    endfor;
endfor;
```

Step 3. For each maximal configuration, check whether it is included in another maximal configuration

```
for all cconf_i∈ cconfs do
    for all cconf_j∈ cconfs (i ≠ j) do
```

if $cconf_i.E \subseteq cconf_j.E$ then

cconfs := cconfs – $cconf_i$;

endif;

endfor;

endfor;

Let us apply the algorithm described above to the labeled event structure of Example 2 (Figure 3.1).

After the first step of the algorithm (identify all maximal configurations of the complete prefix), the maximal configurations are as follows:

(1) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, t5, m4, m5, r4, r5, r6, m11, m12, t10, t11, m17, m18;

(2) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, t5, m4, m5, r4, r5, r6, m11, m12, t10, t11, t12, t13, t14, m19, m20, m21;

(3) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, t5, m6, m7, t7, t8, m14, m15, r7, r8, r11, r13, r12, r14, r15, r16, r20-1, r20-2, r21, r22, r9;

(4) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t6, m8, m9, t9;

After the second step of the algorithm (restrict the events to only those occurring at the PCOs), the configurations are:

1) t1, r2, t3, t5, r5, t11

2) t1, r2, t3, t5, r5, t11, t12, t14

3) t1, r2, t3, t5, t8, r8

4) t1, r2

After the third step of the algorithm (check each maximal configuration to see if it is covered by any other maximal configuration), the configurations are given below:

1) t1, r2, t3, t5, r5, t11, t12, t14

2) t1, r2, t3, t5, t8, r8

3.4 Maximal Configuration Generation for Data Flow Oriented Testing

In our approach, the data flow oriented test generation from the complete prefix of the labeled event structure of a system of asynchronously communicating EFSMs is carried out in the following steps:

1. Identify all maximal configurations to satisfy the all-uses criterion
2. Restrict the events in configurations to only those occurring at the PCOs;
3. For each maximal configuration, check whether it is included in another maximal configuration. If it is, then eliminate it from the set of maximal configurations.

Step 2 is the same as Step 2 of the control flow oriented test generation procedure, and it will not be repeated here. Only steps 1 and 3 will be described in this section in detail.

Step 1 (Identify all maximal configurations to satisfy the all-uses criterion) is carried out by the following sub-steps:

1) Identification of defs and *c*-uses in the labeled event structure

2) Construction of du-pairs with the existing defs and *c*-uses

3) Identification of maximal configurations to cover the du-pairs constructed in step 2

4) Construction of new du-pairs for all the defs, which are not being used in current labeled event structure, by appending a sub-structure containing corresponding *c*-uses

5) Identification of maximal configurations to cover the new du-pairs constructed in Step 4 in the expanded labeled event structure

The algorithm for each sub-step will be discussed in the following sections.

Given a complete prefix $\sigma = \langle E, \preceq, \#, l \rangle$ of a labeled event structure of a system of asynchronously communicating EFSMs *m*, (The complete prefix is represented in the internal data structure defined in 3.1), construct a set of maximal configurations which satisfies the all-uses test selection criterion for data flow oriented testing of the given complete prefix of the labeled event structure.

3.4.1 Identification of defs and c-uses

The algorithm for the classification of variable occurrences in terms of data structures defined for labeled event structure can be outlined as follows. For each action in the action set of a labeled event structure, check the mode of the action:

- If the mode is '?', which indicates that the current action is either a timeout action or an input action, check the parameter list of the action. No variable occurrence is identified if the parameter list is empty. Otherwise, if the first element of the parameter list is a timer identifier (this case only occurs when the action is a timeout action), then a c-use of the timer identifier is identified in this action, the timer identifier is added into the uses field of the current action. Otherwise, the action is an input action and all the elements in the parameter list are regarded as being defined in the action and added into the defs field of the current action.

- If the mode is '!', which indicates that the current action is an output action, check the parameter list of the action. No variable occurrence is

identified if the parameter list is empty. Otherwise, all the elements in the parameter list are regarded as being c-used in the action and added into the uses field of the current action.

- If the mode is '=', which indicates that the current action is an assignment action, the first element of the parameter list is the variable that is being assigned value in the assignment, and thus, it is added into the defs field of the current action; all the other elements of the parameter list are the variables that are being used in the assignment and thus they are added into the uses field of the current action. A special case may happen when one variable is both being used and assigned a value in an assignment statement. In this case, it appears twice in the parameter list.

- If the mode is 's', which indicates that the current action is a timer set action, the first element of the parameter list is the timer identifier which is being set, and thus it is added into the defs field of the current action; all the other elements of the parameter list are the variables that are being used to set the timer identifier and they are added into the uses field of the action.

- If the mode is 'r', which indicates that the current action is a timer reset action, the only element of the parameter list is the timer identifier which is reset, and thus it is added into the uses field of the action.

The algorithm is described in pseudo-code as follows:

Algorithm

For all $act_i \in$ A of the LES

Case (act_i.mode):

'?':

if empty (act_i.paras) then

break;

else if (act_i.paras[1] $\in$ timers) then

act_i.uses := act_i.uses ∪ act_i.paras[1];

else

act_i.defs := act_i.defs ∪ act_i.paras;

endif ;

break ;

'!':

if empty (act_i.paras) then

break;

else

act_i.uses := act_i.uses ∪ act_i.paras;

endif ;

break ;

'=' :

act_i.defs := act_i.defs ∪ act_i.paras[1];

act_i.uses := act_i.uses ∪ act_i.paras[2] ∪ … ∪ act_i.paras[n];

break ;

's' :

act_i.defs := act_i.defs ∪ act_i.paras[1];

act_i.uses := act_i.uses ∪ act_i.paras[2] ∪ … ∪ act_i.paras[n];

break ;

'r' :

act_i.uses := act_i.uses ∪ act_i.paras[1];

break ;

endcase

endfor

By applying the above algorithm, actions in Example 2 (Figure 3.1) are analyzed to contain the following variable occurrences:

Table 3.6 Classification of Variable Occurrences Within Actions of Example 2

action (act_i)	act_i.defs	act_i.uses
t?*.sdatreq(constant,data)	t.data	
t!m.mdatreqDT(data)		t.data

t!*.sdisableind			
t?m.mdatindAK			
t!*.senableind			
m?t.mdatreqDT(data)		m.data	
m!r.mdatindDT(data)			m.data
m?r.mdatreqAK(constant)			
m!t.mdatindAK			
r?m.mdatindDT(data)		r.data	
r!*.sdatind(data)			r.data
r!m.mdatreqAK(constant)			
r:a :=constant		r.a	
r:procedure(pc2(x,y,j;a))	x:=function(a);	r.x	r.a
	y:=function(x);	r.y	r.x
	j:=constant;	r.j	
r:reset(timer2)			r.timer2
r?r.timer1			r.timer1
r:a:=function(a,b)		r.a	r.a, r.b
r:procedure(pc4(x,j;y))	x:=function(j,y);	r.x	r.j, r.y
	j:=constant;	r.j	
r:set(constant,timer3)		r.timer3	
r?r.timer3			r.timer3

3.4.2 Construction of du-pairs

The algorithm for constructing du-pairs in a labeled event structure in terms of data structures defined for a labeled event structure is outlined as follows.

First, initialize the du-pair set. For all *c*-uses of each event in a labeled event structure, find the definition of the same variable in the predecessors of the event occurring at the same component as the event. If there is no definition existing in the same-component predecessors of the current event, continue checking with the same-component predecessors of those predecessors. Stop when either the definition of the same variable is found or the start event of the component is reached. If the definition of the variable is not found, there is a data-flow anomaly (i.e., a use without a preceding def) in the labeled event structure and no du-pair will be created for this *c*-use event.

Once the definition is found, a du-pair is created and inserted into the du-pair set. The du-pair data structure includes the variable, the c-use event, the definition event and the def-clear path, which records all the events that had been checked in the procedure of searching definition event. The definition of the variable in the definition event is marked as "used".

During searching for a timeout or timer reset event, there may be two same-component predecessors: one is the immediate causal predecessor event, the other is the corresponding set event. In such cases, the immediate causal predecessor that is other than the set event should be the only predecessor considered. As described above, the search procedure follows a bottom-up strategy.

The algorithm is described in pseudo-code as follows:

Algorithm

```
dups := ∅;
for all e_i ∈ E do
    for all use_j ∈ e_i.act.uses do
        dcpath := ∅;
        findef := false;
        dcpath := dcpath ∪ e_i;
        predevs := e_i.predecessors ∩ coloc(e_i);
        while (predevs.size > 0) do
            if (predevs.size == 1) then
                predev := predevs[1];
            else // this case only may occur in reset and timeout actions
                for all predev_k ∈ predevs do
                    if (predev_k.act.mode != 's') then
                        predev := predev_k;
                        break;
                    else if (e_i.act.mode == '?') and (e_i.act.msg !=
                    predev_k.act.para[1]) then
```

```
                                predev := predev_k;
                                break;
                        else if (e_i.act.mode == 'r') and (e_i.act.para[1] !=
                        predev_k.act.para[1]) then
                                predev := predev_k;
                                break;
                        endif;
                endfor;
        endif;
        dcpath := dcpath ∪ predev;
        for all def_k ∈ predev.act.defs do
                if (def_k.var == use_j.var) then
                        findef := true;
                        break;
                endif;
        endfor;
        if (findef ) then
                newdup := ⟨ e_i.var, predev, e_i, dcpath ⟩;
                dups := dups ∪ newdup;
                predev.usedefs[def_k] := true;
                break;
        else
                e_i := predev;
                predevs := e_i.predecessors ∩ coloc(e_i);
        endif;
    endwhile;
  endfor;
endfor;
functions:
function coloc(e_i) // returns the subset of events located in the same component as e_i
        evs := ∅;
```

```
    for all e_j ∈ E do
        if (e_j.loc == e_i.loc) then
            evs := evs ∪ e_j;
        endif;
    endfor;
    return evs;
endfunction;
```

By applying the above algorithm, we can tell that Example 2 (Figure 3.1) contains the following du-pairs:

Table 3.7 du-pairs of Example 2

No.	Compo-nent	var	def event	cuse event	def-clear path
1	t	t.data	t1	t2	t1->t2
2	t	t.data	t3	t4	t3->t4
3	t	t.data	t12	t13	t12->t13
4	m	m.data	m1	m2	m1->m2
5	m	m.data	m4	m5	m4->m5
6	m	m.data	m14	m15	m14->m15
7	m	m.data	m20	m21	m20->m21
8	r	r.data	r1	r2	r1->r2
9	r	r.data	r4	r5	r4->r5
10	r	r.data	r7	r8	r7->r8
11	r	r.a	r10	r18-1	r10->r18-1
12	r	r.x	r18-1	r18-2	r18-1->r18-2
13	r	r.timer2	r13	r14	r13->r12->r14
14	r	r.timer1	r12	r15	r12->r14->r15
15	r	r.a	r10	r16	r10->r18-1->r18-2->r18-3->r3->r7->r8->r11->r13->r12->r14->r15->r16
16	r	r.b	r11	r16	r11->r13->r12->r14->r15->r16
17	r	r.y	r18-2	r20-1	r18-2->r18-3->r3->r7->r8->r11->r13->r12->r14->r15->r16->r20-1
18	r	r.j	r18-3	r20-1	r18-3->r3->r7->r8->r11->r13->r12->r14->r15->r16->r20-1
19	r	r.timer3	r21	r22	r21->r22

3.4.3 Coverage of du-pairs

In order to obtain the configurations set to cover all the du-pairs, including both du-pairs existing in the given complete prefix of a labeled event structure and the new du-pairs constructed by appending a sub-structure to the prefix, the following steps have to be applied:

(1) Identification of maximal configurations to cover the du-pairs constructed in 3.4.2

(2) For all the defs, which are not being used the complete prefix of the labeled event structure, appending a sub-structure containing corresponding *c*-uses to construct new du-pairs and identification of maximal configurations to cover the new du-pairs

1. Identification of maximal configurations to cover the du-pairs constructed in 3.4.2

The algorithm for identification of configurations to cover all the du-pairs found in a labeled event structure in terms of data structures defined for labeled event structure is outlined as follows.

First, the data structures are initialized. One configuration will be generated for each du-pair. The component set of each configuration has the same content as the component set of the labeled event structure.

In order to obtain the set of events occurring in the configuration, we need an event queue. The *c*-use event of the du-pair is the first event to be put into the event queue and to the event set of the configuration. Loop while the event queue is not empty, dequeue the first event of the queue and put its predecessors (which do not currently exist in the event set of the configuration) into the event queue and to the event set of the configuration. The event set of the configuration is obtained when the event queue is empty.

In order to build each configuration, all the events in the event set of the configuration are put into an initially empty queue. Loop while the event queue is not empty, dequeue an event from the queue, check the interaction set of its successors with

the event set of the configuration. If all the events in the interaction set are already included in the configuration, then add the event into the configuration. Otherwise, enqueue the event back to the event queue. After a configuration is built, it is added into the data flow oriented configuration set.

Given is again a complete prefix $\sigma = \langle E, \preceq, \#, l \rangle$ of the labeled event structure of a system of asynchronously communicating EFSM's, represented in the internal data structure defined in Section 3.1. The algorithm for the identification of maximal configurations to cover the du-pairs is described in pseudo-code as follows:

Algorithm

```
dconfs := ∅;
for all dup_i ∈ dups do
    dconf_i := ⟨∅,∅,∅,∅⟩;
    cusev := dup_i.usev;
    dconf_i.C := σ.C;
    pred_queue := ∅;
    /// get the complete event set (dconf_j.E) for the configuration
    dconf_i.E := dconf_i.E ∪ cusev;
    put(pred_queue, cusev);
    while not empty(pred_queue) do
        ev := get(pred_queue);
        for all e_k ∈ ev.predecessors do
            if e_k ∉ dconf_i.E then
                dconf_i.E := dconf_i.E ∪ e_k;
                put(pred_queue, e_k);
            endif;
        endfor;
    endwhile;
/// generate a maximal configuration from the event set
    backward_conf := ∅;
```

```
        e_queue := ∅;
        for all e_k ∈ dconf_i.E do
                put(e_queue, e_k);
        endfor;
        while not empty(e_queue) do
                ev := get(e_queue);
                if ( (ev.successors ∩ dconf_i.E) ⊆ backward_conf) then
                        backward_conf := backward_conf ∪ ev;
                else
                        put(e_queue, ev);
                endif
        endwhile
        for all e_k ∈ backward_conf (k values from backward_conf.size()-1 to 0) do
                dconf_i.conf := dconf_i.conf ∪ e_k;
        endfor;
        dconfs := dconfs ∪ dconf_i;
endfor;
```

By applying the above algorithm, we can obtain one configuration to cover each of the du-pairs in Table 3.7 for Example 2. Since the inclusion checking of configurations is not done yet by this stage, some configurations in the configuration set may be inclusions of others. The configurations obtained by the above algorithm are listed below:

(1) t0, t1, t2

(2) t0, t1, t2, t3, t4

(3) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, t5, m4, m5, m11, m12, t10, t11, t12, t13

(4) t0, t1, t2, m0, m1, m2

(5) t0, t1, t2, m0, m1, m2, t3, t4, m4, m5

(6) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, m6, m7, m14, m15

(7) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, t5, m4, m5, m11, m12, t10, t11, t12, t13, m19, m20, m21

(8) t0, t1, t2, m0, m1, m2, r0, r1, r2

(9) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, m4, m5, r4, r5

(10) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, m6, m7, m14, m15, r7, r8

(11) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1

(12) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2

(13) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, m6, m7, m14, m15, r7, r8, r11, r13, r12, r14

(14) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, m6, m7, m14, m15, r7, r8, r11, r13, r12, r14, r15

(15) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, m6, m7, m14, m15, r7, r8, r11, r13, r12, r14, r15, r16

(16) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, m6, m7, m14, m15, r7, r8, r11, r13, r12, r14, r15, r16

(17) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, m6, m7, m14, m15, r7, r8, r11, r13, r12, r14, r15, r16, r20-1

(18) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, m6, m7, m14, m15, r7, r8, r11, r13, r12, r14, r15, r16, r20-1

(19) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, m6, m7, m14, m15, r7, r8, r11, r13, r12, r14, r15, r16, r20-1, r20-2, r21, r22

One configuration is found for each du-pair, and the configuration must cover the def-clear path of the corresponding du-pair. For example, the def-clear path for du-pair 14 is "r12->r14->r15", which is covered by configuration (14) of the above list.

2. For all the defs, which are not being used in the complete prefix of the labeled event structure, appending a sub-structure containing corresponding *c*-uses to construct new du-pairs and identification of maximal configurations to cover the new du-pairs

The algorithm for appending a sub-structure to the complete prefix of the labeled event structure to construct du-pairs to cover the definitions which are not being used in the complete prefix in terms of data structures defined for a labeled event structure is outlined as follows.

Check every event in the event set of the labeled event structure to see whether it has any definition that has not being used. For each un-used definition x of an event, initialize a queue to store intermediate configurations (intermediate configurations are the configurations obtained during the search and appending procedure, before the *c*-use event is found and appended). The configuration that contains the event having the un-used definition x is put into the intermediate configuration queue.

For each configuration taken out of the queue, get its cutoff point and the corresponding point of the cutoff point in the labeled event structure. The corresponding point may appear as a branching point of a labeled event structure and be followed by several sub-structures. We call each configuration that contains the corresponding point as *reference configuration*. The algorithm searches the corresponding *c*-use event along every sub-structure following the corresponding point in each of the reference configuration. (For the search of *c*-use event for one definition, each corresponding point can only be considered once. The corresponding points that had been considered in the search are marked as *visited corresponding points* to avoid being used again.) If another

definition of x is found before the c-use event of x in a reference configuration, the search within this reference configuration terminates. Otherwise, if such a c-use event is found in a reference configuration, then append the part of the reference configuration, from the corresponding point to the c-use event, to the original configuration, and the search terminates successfully. The status of the definition is set to be "used" and the newly obtained configuration is inserted into the data flow oriented configuration set. Otherwise (if neither definition nor c-use of x is found in the reference configuration), append the sub-structure of the reference configuration from the corresponding point until the cutoff point of the reference configuration to the current configuration. If the corresponding point of the cutoff point of the newly obtained configuration is not included in the visited corresponding point set, put the newly obtained configuration into the intermediate configuration queue for further search. The search terminates when either the configuration queue is empty or the status of the definition is changed to "used".

From the description, we can know that at most one configuration can be generated for each un-used definition. In some cases, if no proper c-use is found for a definition, no sub-structure will be appended and no configuration will be generated.

Given is again a complete prefix $\sigma = \langle E, \preceq, \#, l \rangle$ of the labeled event structure of a system of asynchronously communicating EFSM's, represented in the internal data structure defined in Section 3.1. The algorithm is described in pseudo-code as follows:

Algorithm

```
for all e_i∈ σ.E do
    for def_j∈ e_i.act.defs with def_j∉ e_i.usedefs do
        cdpfound := false;
        while (def_j∉ e_i.usedefs) do
            for cconf_k∈ cconfs with e_i∈ cconf_k do
                visited := ∅;
                conf_queue := ∅;
                put(conf_queue, cconf_k);
```

```
        while (not empty (conf_queue) and def_j ∉ e_i.usedefs) do
            appconf := get(conf_queue);
            corresp := appconf.cutoffp.correspoint;
            visited := visited ∪ corresp;
            for cconf_m ∈ cconfs with corresp ∈ cconf_m do
                ddpfoundfirst := false;
                for e_n ∈ cconf_m.conf with cconf_m.conf.index(corresp.coloc(e_n)) <
                cconf_m.conf.index(e_n) do
                    if (def_j.var ∈ e_n.act.defs.vars and def_j.var ∉ e_n.act.uses.vars) then
                        ddpfoundfirst := true;
                        break;
                    else if (def_j.var ∈ e_n.act.uses.vars) then
                        cdpfound := true ;
                        break ;
                    endif ;
                endfor ;
                if (cdpfound) then
                    newconf := subappend(appconf,cconf_m,cconf_m.conf.index(e_n) +
                    1);
                    e_i.usedefs := e_i.usedefs ∪ def_j;
                    dconfs := dconfs ∪ newconf;
                    break;
                else if (!ddpfoundfirst)||(ccomf_m.cutoffp.correspoint ∉ visited) then
                    newconf := subappend(appconf, cconf_m, 0);
                    conf_queue := conf_queue ∪ newconf;
                    visited := visited ∪ corresp;
                endif;
            endfor;
        endwhile;
    endfor;
endwhile;
```

```
    endfor;
endfor;
function subappend:
Configuration subappend(Configuration appconf, Configuration reconf, int cuindex)
/* The first configuration parameter is the configuration to be appended;
The second configuration parameter is the reference configuration, which contains the
corresponding point of the cutoff point of the first configuration
cuindex is the index of cuse event. "0" means no c-use event is found in the reference
configuration, otherwise, it indicates the index of the c-use event in the reference
configuration */
    corresp := appconf.cutoffp.correspoint;
    for all com_w ∈ comp do
        index_w := reconf.conf.index(e_x ∈ corresp and e_x.loc == com_w);
    endfor;
    if (cuindex ≠ 0) then
        for e_v ∈ reconf.conf with (reconf.conf.index(e_v) > index_ev.loc and
        reconf.conf.index(e_v) < cuindex) do
            appconf.conf := appconf.conf ∪ e_v;
        endfor
    else
        for e_v ∈ reconf.conf with (reconf.conf.index(e_v) > index_ev.loc) do
            appconf.conf := appconf.conf ∪ e_v;
            appconf.cutoffp := reconf.cutoffp;
        endfor;
    endif;
    return (appconf);
endfunction;
```

After applying the algorithm of appending a sub-structure to cover the definitions which are not being used in the complete prefix of the labeled event structure, we get new configurations.

For Example 2 (Figure 3.1), there are three definitions that are unused in the complete prefix of the labeled event structure: "r.*a*" at "r16", "r.*x*" at "r20-1" and "r.*j*" at "r20-2".

For the unused definition of "r.*a*" at "r16", the sub-structure is appended from cutoff point "t8, m15, r9" according to the algorithm. First of all, we find the corresponding global state of cutoff point "t8, m15, r9", which is "t2, m2, r3"; then, we look for a *c*-use of variable "r.*a*" in all the sub-structures following global state "t2, m2, r3". In this example, "r16" in sub-structure "t3, t4, m6, m7, m14, m15, r7, r8, r11, r13, r12, r14, r15, r16" contains a *c*-use of variable "r.*a*" and there is no re-definition of "r.*a*" before "r16" in the sub-structure. Finally, we append the above sub-structure to the prefix of labeled event structure after the cutoff point "t8, m15, r9". Thus, we get one more configuration for the unused definition of "r.*a*". The new configuration is listed below:

N1) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, t5, m6, m7, t7, t8, m14, m15, r7, r8, r11, r13, r12, r14, r15, r16, r20-1, r20-2, r21, r22, r9, (t3), (t4), (m6), (m7), (m14), (m15), (r7), (r8), (r11), (r13), (r12), (r14), (r15), (r16)

For the unused definition of "r.*x*" at "r20-1", the sub-structure should be appended from cutoff point "t8, m15, r9" according to the algorithm. First of all, we find the corresponding global state of cutoff point "t8, m15, r9", which is "t2, m2, r3"; then, we look for *c*-use of variable "r.*x*" in all the sub-structures following global state "t2, m2, r3". In this example, no *c*-use of variable "r.*x*" can be found in any of the sub-structures following the corresponding global state "t2, m2, r3". Thus, we cannot get any new configuration for the unused definition of "r.*x*".

For the unused definition of "r.*j*" at "r20-2", the sub-structure is appended from cutoff point "t8, m15, r9" according to the algorithm. First of all, we find the corresponding global state of cutoff point "t8, m15, r9", which is "t2, m2, r3"; then, we look for a *c*-use of variable "r.*j*" in all the sub-structures following global state "t2, m2, r3". In this example, "r20-1" in sub-structure "t3, t4, m6, m7, m14, m15, r7, r8, r11, r13, r12, r14, r15, r16, r20-1" contains a *c*-use of variable "r.*j*" and there is no re-definition of

"r.*j*" before "r20-1" in the sub-structure. Finally, we append the above sub-structure to the prefix of labeled event structure after the cutoff point "t8, m15, r9". Thus, we get one more configuration for the unused definition of "r.*j*". The new configuration is listed below:

N2) t0, t1, t2, m0, m1, m2, r0, r1, r2, r10, r18-1, r18-2, r18-3, r3, t3, t4, t5, m6, m7, t7, t8, m14, m15, r7, r8, r11, r13, r12, r14, r15, r16, r20-1, r20-2, r21, r22, r9, (t3), (t4), (m6), (m7), (m14), (m15), (r7), (r8), (r11), (r13), (r12), (r14), (r15), (r16), (r20-1)

The events whose names are composed of a pair of brackets and an event name mean new events that duplicate some events, whose names are shown inside the brackets, in the complete prefix of the labeled event structure. The new du-pairs created and covered by appending sub-structures are shown in Table 3.8. Since the inclusion checking is not done at this stage, some of the new configurations may be included in others.

Table 3.8 New du-pairs of Example 2, Created and Covered by Appended Sub-structures

No.	Com	var	def event	c-use event	def-clear path
N1	r	r.a	r16	(r16)	r16->r20-1->r20-2->r21->r22->r9- >(r7)->(r8)->(r11)->(r13)->(r12)->(r14)->(r15)->(r16)
N2	r	r.j	r20-2	(r20-1)	r20-2->r21->r22->r9- >(r7)->(r8)->(r11)->(r13)->(r12)->(r14)->(r15)->(r16)->(r20-1)

3.4.4 Elimination of Inclusions

Step 3 of the test generation is inclusion elimination. In the configuration set for data flow oriented testing, there is one configuration generated for each du-pair. Some configurations may be included in others. The elimination of such inclusions is needed to obtain the final set of configurations. The algorithm to eliminate included configurations from the data flow oriented configuration set is composed of two steps: the first step is

the same as the procedure of inclusion elimination in control flow oriented configuration set; the second step considers the duplicate events in the appended sub-structures. In this step, to check if a configuration is included in another, two criteria have to be examined: firstly, if the size of a configuration is greater than the other, the former one is not included in the latter one; secondly, if any event of a configuration have different actions with the same-index event of the second configuration, the former configuration is not included in the latter one.

The algorithm is described in pseudo-code as follows:

Algorithm

```
for all cconf_i ∈ cconfs do
    for all cconf_j ∈ cconfs (i ≠ j) do
        if (cconf_i.coveredby(cconf_j)) then
            cconfs := cconfs – cconf_i;
        endif;
    endfor;
endfor;
Function coveredby:
bool Configuration::coveredby(Configuration* cfj)
    if (this.conf.size > cfj.conf.size) then
        return false;
    endif;
    for all e_i ∈ this.conf do
        if (e_i != cfj.conf.event(i)) then
            return false;
        endif;
    endfor;
    return true;
endfunction;
```

Before applying the algorithm of events restriction and elimination of inclusions, the data flow oriented configuration set contains twenty-one (nineteen for existing du-pairs in the labeled event structure and two for new du-pairs constructed for definitions without *c*-uses (both listed in 3.4.3)) configurations for Example 2.

After applying the algorithm of restriction of the events to only those occurring at the PCOs, the configurations are as follows:

(1) t1

(2) t1, t3

(3) t1, r2, t3, t5, t11, t12

(4) t1

(5) t1, t3

(6) t1, r2, t3

(7) t1, r2, t3, t5, t11, t12

(8) t1, r2

(9) t1, r2, t3, r5

(10) t1, r2, t3, r8

(11) t1, r2

(12) t1, r2

(13) t1, r2, t3, r8

(14) t1, r2, t3, r8

(15) t1, r2, t3, r8

(16) t1, r2, t3, r8

(17) t1, r2, t3, r8

(18) t1, r2, t3, r8

(19) t1, r2, t3, r8

(N1) t1, r2, t3, t5, t8, r8, (t3), (r8)

(N2) t1, r2, t3, t5, t8, r8, (t3), (r8)

After applying the algorithm of elimination of inclusions, the configuration set is reduced to three configurations. The set of configurations are listed below. Table 3.9 shows the inclusions of each configuration.

7) t1, r2, t3, t5, t11, t12

9) t1, r2, t3, r5

N2) t1, r2, t3, t5, t8, r8, (t3), (r8)

Table 3.9 Inclusions of Each Configuration

Configuration No.	Included by configuration
1	7
2	7
3	7
4	7
5	7
6	7
7	Not included in any configuration
8	7
9	Not included in any configuration
10	N2
11	N2
12	N2
13	N2
14	N2
15	N2
16	N2
17	N2

18	N2
19	N2
N1	N2
N2	Not included in any configuration

3.5 Changes with respect to the Original Specification

In [1], the labeled event structure is generated from an expanded specification of the original system of asynchronously communicating EFSMs, such that in the expanded specification, there are no control variables. The details of the transformation from a system of asynchronously communicating EFSMs to an expanded specification are described in [1]. This transformation leads to the removal of control variables and an increase in the number of states and input signals. The control variables are eliminated by shifting them to a set of new states or to a set of new inputs. A new input is a combination of the original input and a concrete assignment of values to all parameters associated with this input that are control variables.

For example, as shown in Figure 2.1, the expanded specification of Example 1, module "m" has input signals "mdatindDT" and "mdatindAK", they are compound signals that are constructed by joining the original input "mdatind" and concrete assignment of values, "DT" and "AK", which are associated with a control variable in the original specification.

In our approach, after the test configurations are generated by our algorithm, the compound signal names in the configurations should be changed back to be consistent with the signals in the original specification, and the concrete assignments of values in the new inputs are used to be the test data for test cases.

3.6 Test Data Selection

Variables in the original specification can be classified into two classes: control variables and non-control variables.

In the specification transformation procedure, the values that can be assigned to control variables are integrated with the inputs. Thus, in the test data selection, those values should be selected as the test data for the test, where the composed inputs exist.

Non-control variables are left over after the transformation. They occur in the send events of the test suite. The test suite designer has to choose a suitable value for it. What a "suitable" value means, is up to the test suite designer. The variables remaining in the test suite are defined as test suite variables.

Chapter 4 A System Prototype

A prototype of the approach described in this thesis is implemented in C++ language and runs on Sun workstations under Solaris 5.8. The objective of the prototype is to provide a tool that can automatically generate an executable test suite from a labeled event structure representation of a system of asynchronously communicating EFSMs. Based on the automatic test suite generation approach proposed in this thesis, the prototype consists of three major parts, which are shown in Figure 4.1:

Phase 1: Processing a labeled event structure

Phase 2: Configuration Construction

Phase 3: Test Construction

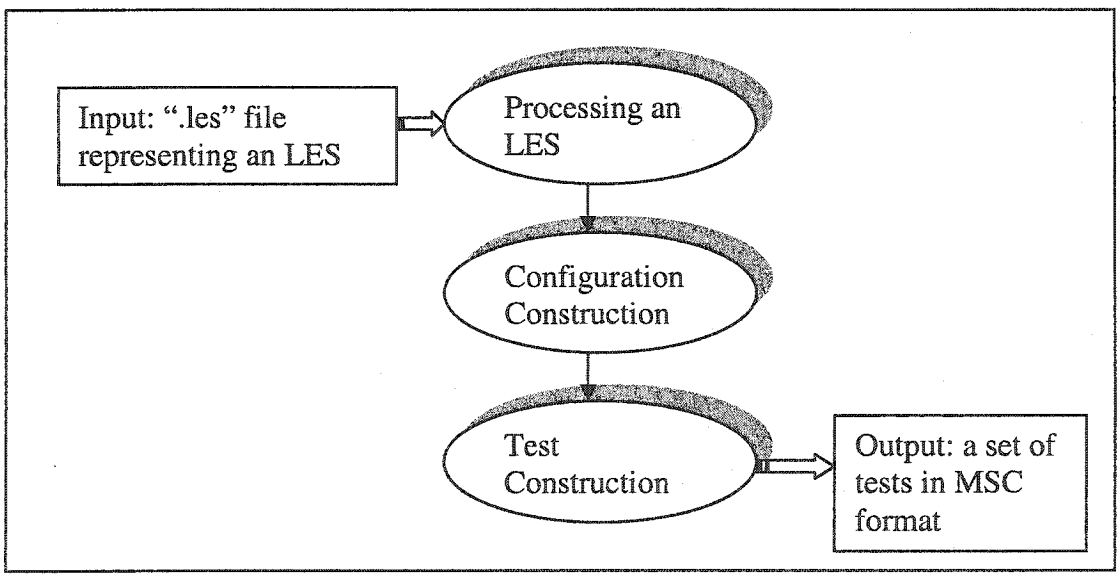


Figure 4.1 The Software Configuration

The process performed by the software can be described as follows. First, a ".les" file representing a labeled event structure is analyzed by the parser and the information on the labeled event structure is filled into the internal data structure. Second, based on the labeled event structure information in the internal data structure, a set of configurations is constructed for control flow oriented testing or data flow oriented

testing. Third, the configurations are put in textual MSC format. There is some post-processing work that has to be done manually: for each MSC representing a configuration, signal names are changed back to the original signal names according to the original specification of the system of asynchronously communicating EFSMs, and test data is selected for each signal parameter. Thus, the test suite for the system of asynchronously communicating EFSMs is obtained.

In section 4.1, the input file format of the prototype is given. The output file format of the prototype is given in section 4.2. The implementations of Phase1, Phase 2 and Phase 3 are discussed in detail in sections 4.3, 4.4 and 4.5, respectively.

4.1 Input File Format

The input file representing a labeled event structure in this thesis is named as ".les" file. ".les" file is a quintuple $\langle E, S, \preceq, \#, C \rangle$, where:

- E is a countable set of labeled events.
  Each event has the following format:
  'label(' eventid ')' '=' action ';'.
  An example of an event in E in Example 2 is
  "label(t2)=t!m.mdatreqDT(updu);"

- S is a set of start points, one start point for each component.
  Each start point has the following format:
  'root' eventid ';'.
  An example of a start point in S in Example 2 is:
  "root et0;".

- $\preceq \subseteq E \times E$ is a partial order representing the causality relation.
  It has the following format:
  eventid '<' ... '<' eventid ';'.
  An example of causally related events in Example 2 is:

"t0<t1<t2;".

- # $\subseteq$ E ×E is the conflict relation.

It has the following format:

eventid '#' eventid ';'.

An example of conflicting events in Example 2 is:

"t3#t6;".

- *C* is a set of cutoff point/global state pairs representing the relationships between cutoff points and their corresponding points.

A cutoff point/global state pair has the following format:

'{' cutoff point '}' '->' '{' corresponding point '}' ';'.

An example of a cutoff point/global state pair in Example 2 is:

"{t9, m9, r3} -> {t0, m0, r0};". In this example, the corresponding global state of cutoff point "t9, m9, r3" is "t0, m0, r0".

The detailed definition of ".les" file in extended Backus-Naur Form is as follows:

```
les          ::= labels starts relations cutoffs .
labels       ::= {label} .
label        ::= 'label' '(' eventid ')' '=' action ';' .
action       ::= input
                 | output
                 | assignment
                 | set
                 | reset
                 | procedure_call .
input        ::= Id ':' Id '?' Id '.' Id ['(' parameters ')'] .
output       ::= Id ':' Id '!' Id '.' Id ['(' parameters ')'] .
assignment   ::= Id ':' variableid ':=' expression .
set          ::= Id ':' 'set' '(' 'constant' ',' timer ')' .
```

```
timer          ::= Id .
reset          ::= Id ':' 'reset' '(' timer ')' .
procedure_call ::= Id ':' 'procedure' '(' procedureid '(' [variableids] ')' ')' ['{' pbrdefs
'}'].
procedureid    ::= Id .
parameters     ::= parameter {',' parameter} .
parameter      ::= variableid
                 | 'constant' .
variableids    ::= variableid{',' variableid} .
variableid     ::= Id .
expressions    ::= expression{',' expression} .
expression     ::= 'function' '(' variableids ')'
                 | 'constant' .
pbrdefs        ::= {pbrdef}.
pbrdef         ::=  variableid ':=' expression ';' .
starts         ::= {start_point} .
start_point    ::= 'root' eventid ';' .
eventid        ::= Id .
relations      ::= {relation} .
relation       ::= causality
                 | conflict .
causality      ::= eventid '<' eventid ';'
                 | eventid '<' causality .
conflict       ::= eventid '#' eventid ';' .
cutoffs        ::= {cutoff} .
cutoff         ::= '{' eventids '}' '->' '{' eventids '}' ';' .
eventids       ::= eventid{',' eventid} .
Id             ::= identifier .
```

An example ".les" file for Example 2 is given in Appendix B.1.

4.2 Output File Format

The output of the prototype is a set of MSC (Message Sequence Chart) files, which can be accepted by Telelogic Tau. The intent is to represent each test case in TTCN.

A message sequence chart (MSC) is a high-level description of the message exchange between system components and with the system components and their environment. A major advantage of the MSC language is its clear and unambiguous graphical layout which immediately gives an intuitive understanding of the described system behavior [27]. The syntax and semantics of MSCs are standardized by ITU-T, as recommendation Z.120 [30].

Basically, an MSC describes the information interchange which is carried out by sending and receiving messages among processes. In an MSC, these messages would coincide with the signals that are sent from one process and consumed by another process. The processes would correspond to components of the system of the asynchronously communicating EFSMs or the environment the system is in [27].

The most fundamental language constructs of MSCs are instances (e.g., processes and environment services) and messages describing the communication events. Messages exchanged between the processes and environment services are represented by the message names and parameters (if there are any). Besides the input and output messages, there are also some other events that can be recognized in MSC, e.g., timers and tasks, which are executed within a process.

Each event causes the insertion point to be translated downwards with one vertical spacing unit, keeping the intuitive feeling of *absolute order* between events. In our approach, some of these absolute orders have to be followed strictly when generating the test cases: the absolute order of message interchange events (input and output) has to be followed, since the messages has to be received by a process after it is sent by another process; and the absolute order of events within one process has to be strictly followed. Both of these two types of orders come from the causality relation of the algorithm for constructing prefix of a labeled event structure. On the other hand, ordering of events

occurring concurrently does not need to be followed strictly. The ordering of the concurrent events is arbitrary because in these cases the algorithm arbitrarily chooses events of one EFSM to be appended to the labeled event structure.

An MSC example is given in Figure 4.2

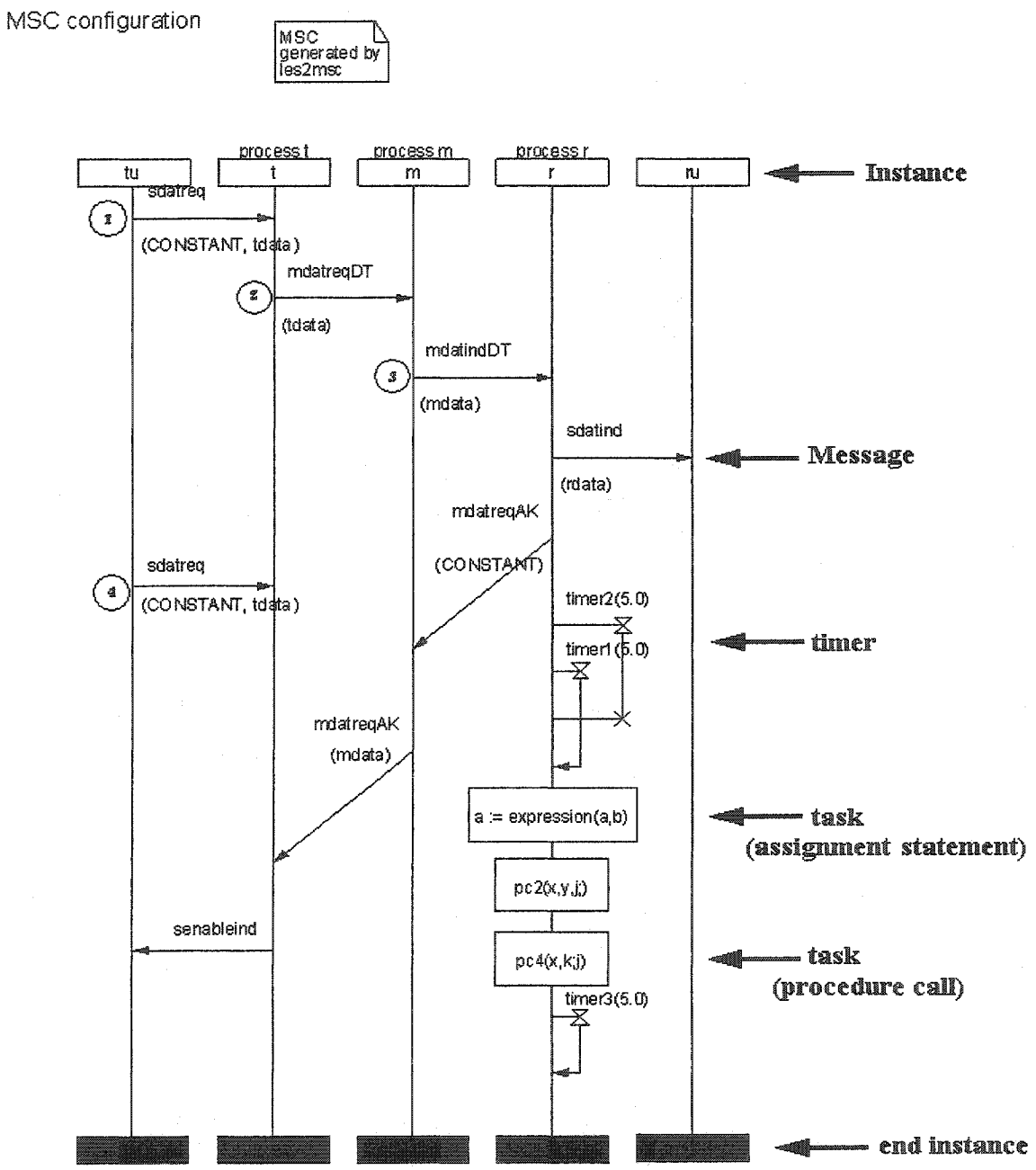


Figure 4.2 An MSC Example

As shown in Figure 4.2, the example system has three processes (named as "r", "m" and "t") and two environment services (named as "tu" and "ru"). In the message interchange marked with (1), message "sdatreq(CONSTANT, data)" is sent by environment service "tu" and received by process "t". In the message interchange marked with (2), message "mdatreqDT(data)" is sent by process "t" and received by process "m". Timers "timer1" and "timer2" occur within process "r". Tasks (e.g., assignment "a:=expression(a,b)" and procedure "pc2(x,y,j;a)") occur within process "r".

The message interchange (2) must occur after the message interchange (1), because process "t" has to receive the message "sdatreq(CONSTANT, data)" from environment service "tu" before sending out message "mdatreqDT(data)" to process "m". There is no strict order of occurrence for the message interchange (3) and the message interchange (4), since they have no causality relationship, and thus can occur in parallel or in any order.

The output MSC files describe the set of test cases. The sets of MSCs generated by the prototype for Example 2 are given in Appendix A and the sets of MSCs for Example 3 are given in Appendix B.

The constructed MSCs can be validated against the original system of asynchronously communicating EFSMs and be used to check the behavior of the system under test. Telelogic Tau supplies a tool named Autolink to generate TTCN test cases from system level MSCs.

4.3 Phase 1: Processing a Labeled Event Structure

Phase 1 (Processing a labeled event structure) converts an input (a labeled event structure file) into an internal data structure defined in Chapter 3.1, containing the information of the labeled event structure, including the events, their predecessors and successors, the data dependencies associated with the actions as well as the cutoff points with their corresponding points.

In the prototype tool, Lex and Yacc are used to get the information of the labeled event structure from the input ".les" file, and put it into the internal data structure. Lex is a tool to generate lexical analyzers. The code generated by Lex is used to read input characters and produce as output a sequence of tokens that a parser uses for syntax analysis. Yacc is a tool to generate parsers. The code generated by Yacc is used to get tokens from the lexical analyzer and to fill the information of the labeled event structure into the internal data structure according to the grammar. The roles of lexical analyzer and parser in a compiler are shown in Figure 4.3.

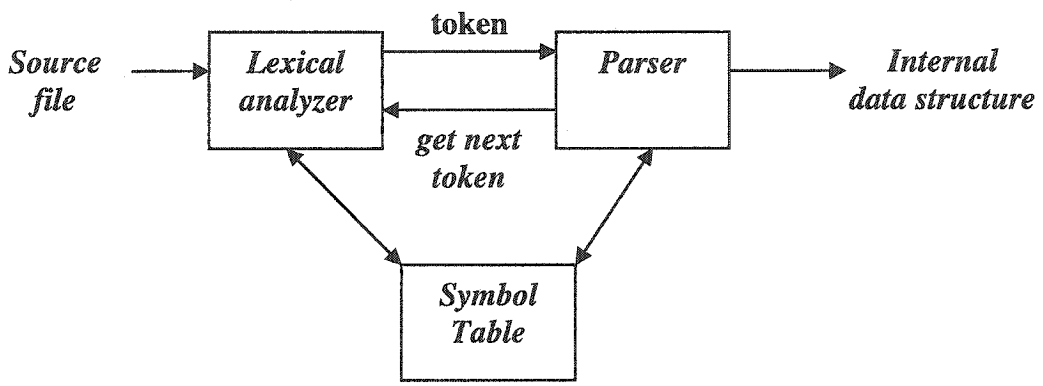


Figure 4.3 Interaction of Lexical Analyzer with Parser

Table 4.1 Keywords of the Prototype Used in the Lexical Analyzer:

Keywords	Usage in labeled event structures
LABEL	events
ROOT	root node
SET	set events
RESET	reset events
PROCEDURE	procedure call events
FUNCTION	expressions containing variables
CONSTANT	expressions without variables

Other tokens used in the prototype are listed in Table 4.2:

Table 4.2 Tokens of the Prototype Used in Lexical Analyzer

Tokens	Usage in labeled event structures
Digit	[0-9]
Letter	[a-zA-Z_]
Id	{Letter}({Letter}|{Digit})*

Whitespace	[\t\f]*
/*	comments start
*/	comments end
\n	line separator
->	corresponding global states
(	(
)	)
{	{
}	}
:	:
=	=
,	,
;	;
.	.
!	!, output event
?	?, input event
<	<, event causality
#	#, event conflict

The lexical analyzer developed with Lex is available at www.site.uottawa.ca/~ural/. Examples of an event, a conflict relationship and a cutoff point/global state pair are given below:

Table 4.3 Lexeme and Tokens for Event "label(t12)=t?tu.sdatreq(constant,data);"

Lexeme	Tokens
label	LABEL
(	(
t12	Id
)	)
=	=
t	Id
?	?
tu	Id
.	.
sdatreq	Id
(	(
constant	CONSTANT
,	,
data	Id
)	)
;	;

According to the grammar given in Section 4.1, these tokens construct of a legal event, which is acceptable by the parser.

Table 4.4 Lexeme and Tokens for Conflict Relationship of Events "t6#m6;"

Lexeme	Tokens
t6	Id
#	#
m6	Id
;	;

According to the grammar, these tokens construct of a legal event conflict relationship between event "t6" and "m6".

Table 4.5 Lexeme and Tokens for Cutoff point/global state pair "{t9, m9, r3}->{t0, m0, r0};"

Lexeme	Tokens
{	{
t9	Id
,	,
m9	Id
,	,
r3	Id
}	}
->	->
{	{
t0	Id
,	,
m0	Id
,	,
r0	Id
}	}
;	;

According to the grammar, these tokens construct of a legal pair between global state "t9, m9, r3" and "t0, m0, r0".

The parser of this prototype is developed with Yacc. It gets tokens from the lexical analyzer, and puts the information into the internal data structures defined in Section 3.1. The parser is available at "www.site.uottawa.ca/ural".

To identify the variable names in different component of the system of asynchronously communicating EFSMs, the component name with a dot is prepended to each variable. For example, variable *data* in component *t* is changed to *t.data* in the

internal data structure. The following tables are part of the internal data structure after the syntax analysis of Example 2:

Table 4.6 Part of Internal Data Structure of Example 2

- **Actions**

Action	loc	mode	rem	msg	para
t?m.mdatreqDT(data);	t	?	m	mdatreqDT	t.data
m!r.mdatindDT(data);	m	!	r	mdatindDT	m.data
m!t.mdatindAK;	m	!	t	mdatindAK	∅
r:a:=function(a,b);	r	=	r	∅	r.a, r.a, r.b
r:set(constant,timer1);	r	s	r	∅	r.timer1
r:reset(timer2);	r	r	r	∅	r.timer2
r:procedure(pc2(x,y,j;a))	r	=	r	∅	r.x, r.a
{x:=function(a);y:=function(x);	r	=	r	∅	r.y, r.x
j :=constant;};	r	=	r	∅	r.j

- **Events**

Event Id	Action	predecessors	successors
t0	t?t.start	∅	t1
m1	m?t.mdatreqDT(data)	m0, t2	m2
m2	m!r.mdatindDT(data)	m1	m4, m6, m8, r1
t9	t?m.mdatindAK	t6, m9	∅
r6	r!m.mdatreqAK(constant)	r5	m17
r14	r:reset(timer2)	r12, r13	r15

- **Start points**

Start points	t0, m0, r0

- **Cutoff points**

Cutoff point	Corresponding global state
t9, m9, r3	t0, m0, r0
t8, m15, r9	t2, m2, r3
t11, m18, r6	t6, m9, r3
t14, m21, r6	t5, m5, r3

4.4 Phase 2: Configuration Construction

Phase 2 (Configuration Construction) takes the internal data structure constructed in Phase 1 and generates a set of configurations based on the selected test selection criterion for control flow or data flow oriented testing.

Phase 2 consists of two modules: construction of configurations for control flow oriented testing; and construction of configurations for data flow oriented testing. At present, the set of configurations constructed by the prototype for control flow oriented testing satisfies all-nodes test selection criterion, and the set of configurations for data flow oriented testing satisfies all-uses test selection criterion. Since no path predicate exists in the labeled event structure, all the configurations obtained are feasible.

4.4.1 Control Flow Oriented Configuration Construction

The system flow of the module of configurations construction for control flow oriented testing is illustrated in Figure 4.4.

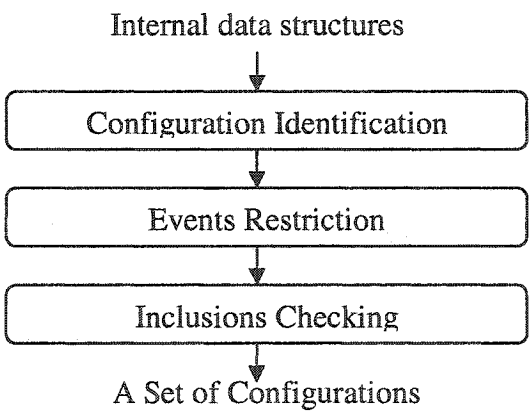


Figure 4.4 System Flow for Control Flow Oriented Test Generation

A short description for each module is given below. The algorithms to implement the functionality of each module are given in Chapter 3.

• **Configuration Identification**

This module is responsible for generating configurations from the labeled event structure. The process of generating a configuration proceeds in a bottom-up fashion. In other words, each configuration is generated from a cutoff point of the labeled event structure, continued by adding predecessors of the events of the cutoff point as well as the predecessors of the predecessors into the configuration, until all the start points of the labeled event structure are reached. Since the configuration construction starts from the cutoff point of the prefix of the labeled event structure, and ends up at the start points of the labeled event structure, all the obtained configurations are maximal configurations of the prefix. For every cutoff point of the prefix, there will be one and only one configuration generated.

After all the events occurring in a configuration are obtained, the sequences of the events in the configuration can be obtained by ordering events in a strategy that all the successors of an event must occur after the event.

The output of this module is a set of configurations in terms of the internal data structures defined in Chapter 3.1. Each configuration by far includes not only entity events (e.g., input, output, assignment, set, reset, timeout), but also other events (i.e., start point of each EFSM and non-observable event τ).

In our approach, the set of configurations for control flow oriented testing covers all the configurations in the labeled event structure and each maximal configuration in the labeled event structure has a corresponding configuration in the obtained set of configurations. Thus, the set of configurations satisfies the all-nodes test selection criterion.

• **Events Restriction**

Since only the events that occur at the PCOs can be observed during testing, all other events in the configurations have to be omitted. This process is done by checking the names of the component and the remote component of events and the events whose

component and the remote component are both not the environment of the system under test, are omitted.

The events domain of the internal data structures is as follows:

- start points. The component name and the remote component name of a start point are the same and they must be a process name in the system of asynchronously communicating EFSMs. Thus, all the start points are omitted in the configuration by this module.

- non-observable event τ. The component name and the remote component name of a non-observable event are the same and they must be a process name in the system of asynchronously communicating EFSMs. Thus, all the non-observable events are omitted from the configuration by this module.

- input. The component name and the remote component name of an input event are different names. They can be the process names in the system of asynchronously communicating EFSMs and/or the environment service names where the system under test is in. Only the events, where one of these names is the name of an environment service, are the events occurring at PCOs, and only they can be observed during testing. Thus, all other events where both of these two names are process names in the system, are omitted from the configuration by this module.

- output. The component name and the remote component name of an output event are different names. They can be the process names in the system of asynchronously communicating EFSMs and/or the environment service names where the system under test is in. Only the events, where one of these names is the name of an environment service, are the events occurring at PCOs, and only they can be observed during testing. Thus, all other events where both of these two names are process names in the system, are omitted from the configuration by this module.

- assignment. The component name and the remote component name of an assignment are the same and they must be a process name in the system of asynchronously communicating EFSMs. Thus, all the assignments are omitted from the configuration by this module.

- set. The component name and the remote component name of a set event are the same and they must be a process name in the system of asynchronously communicating EFSMs. Thus, all the set events are omitted from the configuration by this module.

- reset. The component name and the remote component name of a reset event are the same and they must be a process name in the system of asynchronously communicating EFSMs. Thus, all the reset events are omitted from the configuration by this module.

- timeout. The component name and the remote component name of a timeout event are the same and they must be a process name in the system of asynchronously communicating EFSMs. Thus, all the timeout events are omitted from the configuration by this module.

The output of this module is a set of configurations in terms of the data structures defined in Chapter 3.1. Each configuration only includes the events occurring at PCOs (In the case of our prototype, only part of input and output events of the system of asynchronously communicating EFSMs).

• **Inclusions Checking**

This module checks all the configurations obtained above, to see whether any of them is included in others. All the configurations which are included in some other configuration are removed from the set of configurations.

We say configuration $c1$ is *included in* configuration $c2$ if all the events in $c1$ also occur in $c2$. Then, $c1$ has to be removed from the final set of configurations.

The output of this module is a minimal set of maximal configurations for control flow oriented test generation. They can be translated to MSCs by the prototype, which are the set of tests for control flow oriented testing from the system of asynchronously communicating EFSMs.

4.4.2 Data Flow Oriented Configuration Construction

The system flow of the module for construction of configurations for data flow testing is illustrated in Figure 4.5.

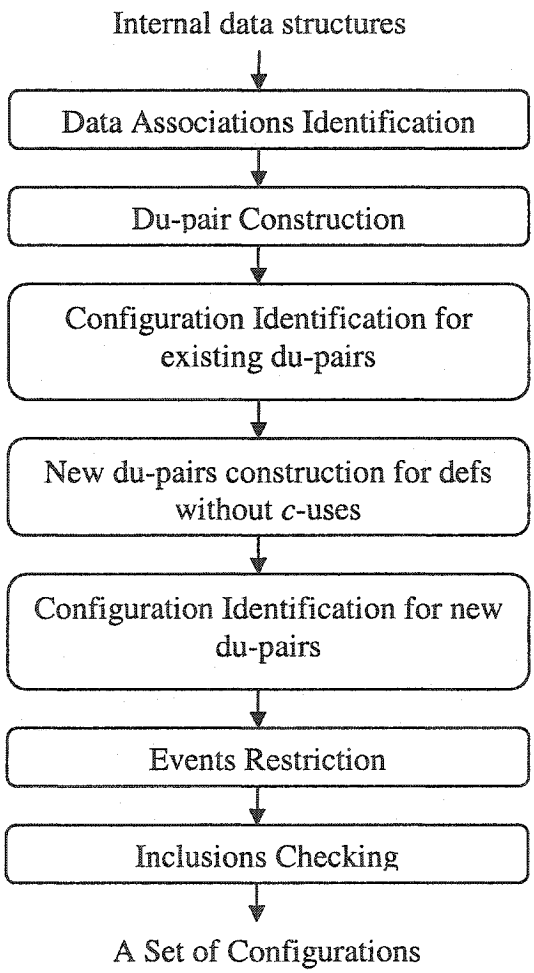


Figure 4.5 System Flow for Data Flow Oriented Test Generation

The modules of "Events Restriction" and "Inclusions Checking" in the procedure of data flow oriented configuration generation are the same as those in the procedure of

control flow oriented configuration generation. A short description for all the other modules is given below.

- **Data Associations Identification**

This module is responsible for identifying definitions and *c*-uses in each action of the labeled event structure. Different mode of actions has different regulations to identify the data associations: input action specifies all its parameters as definitions; output action specifies all its parameters as *c*-uses; assignment action specifies the first parameter as a definition and all the other parameters as *c*-uses; set action specifies the first parameter (a timer identifier) as a definition and all the other parameters as *c*-uses; reset action specifies the only parameter (a timer identifier) as a *c*-use; timeout action specifies the message (a timer identifier) as a *c*-use; procedure call action is broken down into several assignment actions according to the abstract definition of the called procedure and each assignment action is interpreted as above. The information of definitions and *c*-uses identified by this module are put into corresponding fields of the internal data structure of the corresponding action.

The output of this module is a set of actions in terms of the internal data structure defined in Chapter 3.1. Both the list of definitions and the list of *c*-uses fields of the actions are filled up with computed data associations.

- **Du-pair Construction**

Du-pairs are constructed using the information of data associations obtained from the previous module. This module constructs du-pairs along the configurations of the labeled event structure. The construction of each du-pair starts from the *c*-use event, searches corresponding definition among predecessors of the *c*-use event as well as predecessors of the predecessors. Since only local variables are considered in our prototype, the corresponding definition should be found within the same component. Thus, only the predecessors, which occur at the same component as the *c*-use event, are considered during the search. Once the definition event is found, the definition event, the *c*-use event, and all the events that have been traversed between them without

encountering another definition of the same variable form a def-clear path with respect to the variable of the du-pair.

Every *c*-use in the labeled event structure is considered by this module. Ideally, every *c*-use event has a corresponding definition along the configurations of the labeled event structure and a du-pair is obtained accordingly. Although it is very unlikely to happen that in some poorly structured specification, some variables are *c*-used without being previously defined, our prototype still considers this case and forces the search to stop when the start point of the component, where the c-use event occurs, is reached. In this case, no du-pair is constructed for the *c*-use.

The output of this module is a set of du-pairs in terms of the internal data structures defined in Chapter 3.1.

• **Configuration Identification for existing du-pairs**

This module is responsible for identifying configuration for each du-pair that is constructed by the previous module. For each du-pair, the procedure of identification of the events belonging to the configuration starts from the *c*-use event, covers all the predecessors of the *c*-use event as well as predecessors of the predecessors (these predecessors include all the events in the def-clear path of the du-pair and the definition event), ends when all the start points of the labeled event structure have been reached. After all the events in a configuration are identified, the execution sequence of these events is obtained by ordering the events in a strategy that all the successors of an event must occur after the event.

The output of this module is a set of configurations in terms of the internal data structures defined in Chapter 3.1. All the *c*-uses in the labeled event structure are considered to be covered in the obtained set of configurations, while not all the definitions are ensured to be *c*-used yet by this stage.

• **New du-pairs construction for defs without *c*-uses**

Some definitions in the labeled event structures may not have been *c*-used in any du-pair constructed in the module of "du-pair Construction". In order to cover these un-used definitions, sub-structures, which contain *c*-use events for those variables, need to be appended to the labeled event structure.

In our prototype, all the definitions in the labeled event structure are considered. When a definition is found that is not used yet, a procedure of sub-structure appending is started for it. The procedure of sub-structure appending uses the information of cutoff point/global state pairs. When an un-used definition is found in a configuration (we call it current configuration), the prototype will look for the corresponding *c*-use event in the sub-structure following the corresponding global state in some other configurations (we call it reference configuration). If the corresponding *c*-use event is found, the sub-structure of the reference configuration from the corresponding global state to the *c*-use event is appended to the labeled event structure at the cutoff point of the current configuration. Thus the un-used definition, the appended *c*-use event and all the events that have been traversed between them without encountering another definition of the same variable form a new du-pair. Otherwise (if a corresponding *c*-use event is not found in the sub-structure of reference configuration from the corresponding global state until its cutoff point), the sub-structure of the reference configuration from the corresponding global state to its cutoff point is appended to the labeled event structure at the cutoff point of the current configuration, and the appending is repeated from the cutoff point of the newly obtained configuration. In the procedure of sub-structure appending for one un-used definition, each involved corresponding global state is considered only once in order to guarantee the termination of this process.

The output of this module is a set of du-pairs in terms of the internal data structures defined in Chapter 3.1.

• **Configuration Identification for new du-pairs**

A new configuration is identified for every new du-pair constructed by the previous module. The procedure of identification of configuration is the same as the procedure described in module "Configuration Identification for du-pairs".

The output of this module is a set of configurations in terms of the internal data structures defined in Chapter 3.1. All the definitions in the labeled event structure are considered to be covered in the newly obtained set of configurations.

The set of configurations obtained in the module of "Configuration Identification for existing du-pairs" covers all the *c*-uses of the labeled event structure, and the set of configurations obtained in the module of "Configuration Identification for new du-pairs" covers all the definitions of the labeled event structure. The prototype combines these two sets together, and thus, the combined set of configurations covers all the data associates of the labeled event structure. The combined set of configurations satisfies the all-uses test selection criterion.

So far, each configuration in the combined set of configurations includes not only entity events (e.g., input, output, assignment, set, reset, timeout), but also other events (i.e., start point of each EFSM and non-observable event τ), and some configurations may be included in the others. From this combined set of configurations, the prototype continues with the configuration construction phase by the next two steps: Events Restriction and Inclusions Checking which are already given in Section 4.4.1.

After all the modules in the system flow of configurations construction for data flow oriented testing are executed, a minimal set of maximal configurations for data flow oriented testing is obtained. They can be translated to MSCs by the prototype, which are the set of tests for data flow oriented testing from the system of asynchronously communicating EFSMs.

The set of configurations for data flow oriented testing is different than the set of configurations for control flow oriented testing due to the sub-structure appending. One configuration in the labeled event structure may have different representations in the two

set. For example, the third left-most configuration in Figure 3.1 is represented in the set of control flow configuration as "t1, r2, t3, t5, t8, r8" (we name it "*c*1"), and is represented in the set of data flow configuration as "t1, r2, t3, t5, t8, r8, (t3), (r8)" (we name it "*d*1"). "*d*1" has two more message interchanges than "*c*1", because a sub-structure is appended to the labeled event structure to include the test for the un-used definition for variable "r.*a*" at event r16 and variable "r.*j*" at event r20-2.

4.5 Phase 3: Test Construction

Phase 3 (Test Construction) takes the set of configurations from the internal data structure, prints out a set of Message Sequence Charts, which indicates the sequences that the test should follow; modifies signal names according to the original specification; and selects the test data. At present, Phase 3 consists of three steps:

- Form and print out MSCs
- Change signal names with respect to the original specification
- Select test data

The prototype has implemented the function of the first step, and the last two steps have to be done manually.

This module outputs tests in textual MSC format. The textual MSC file format can be recognized by Telelogic Tau and can be shown in graphical MSC format. The textual MSC of an example for control flow oriented test ("*c*1" of Example 2) is given below and Figure 4.6 shows the graphic format of it in Telelogic Tau. This MSC is corresponding to the configuration, whose cutoff point is "t8, m15, r9".

```
msc configuration;
tu: instancehead;
swpx: instancehead;
ru: instancehead;
```

```
tu: out sdatreq,1(CONSTANT, t.data) to swpx;
swpx: in sdatreq,1(CONSTANT, t.data) from tu;
swpx: out sdatind,2(r.data) to ru;
ru: in sdatind,2(r.data) from swpx;
tu: out sdatreq,3(CONSTANT, t.data) to swpx;
swpx: in sdatreq,3(CONSTANT, t.data) from tu;
swpx: out sdisableind,4 to tu;
tu: in sdisableind,4 from swpx;
swpx: out senableind,5 to tu;
tu: in senableind,5 from swpx;
swpx: out sdatind,6(r.data) to ru;
ru: in sdatind,6(r.data) from swpx;
tu: endinstance;
swpx: endinstance;
ru: endinstance;
endmsc;
```

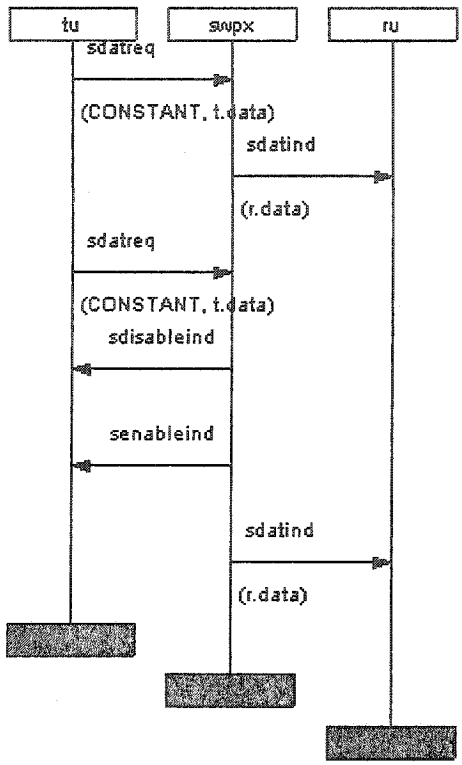


Figure 4.6 A Control Flow Oriented Configuration for Example 2

Configuration "*d*1" is generated for data flow oriented testing. Its textual MSC format is given below and the graphic format in Telelogic Tau is given in Figure 4.7.

msc configuration;

tu: instancehead;

swpx: instancehead;

ru: instancehead;

tu: out sdatreq,1(CONSTANT, t.data) to swpx;

swpx: in sdatreq,1(CONSTANT, t.data) from tu;

swpx: out sdatind,2(r.data) to ru;

ru: in sdatind,2(r.data) from swpx;

tu: out sdatreq,3(CONSTANT, t.data) to swpx;

swpx: in sdatreq,3(CONSTANT, t.data) from tu;

swpx: out sdisableind,4 to tu;

tu: in sdisableind,4 from swpx;

swpx: out senableind,5 to tu;

tu: in senableind,5 from swpx;

swpx: out sdatind,6(r.data) to ru;

ru: in sdatind,6(r.data) from swpx;

tu: out sdatreq,7(CONSTANT, t.data) to swpx;

swpx: in sdatreq,7(CONSTANT, t.data) from tu;

swpx: out sdatind,8(r.data) to ru;

ru: in sdatind,8(r.data) from swpx;

tu: endinstance;

swpx: endinstance;

ru: endinstance;

endmsc;

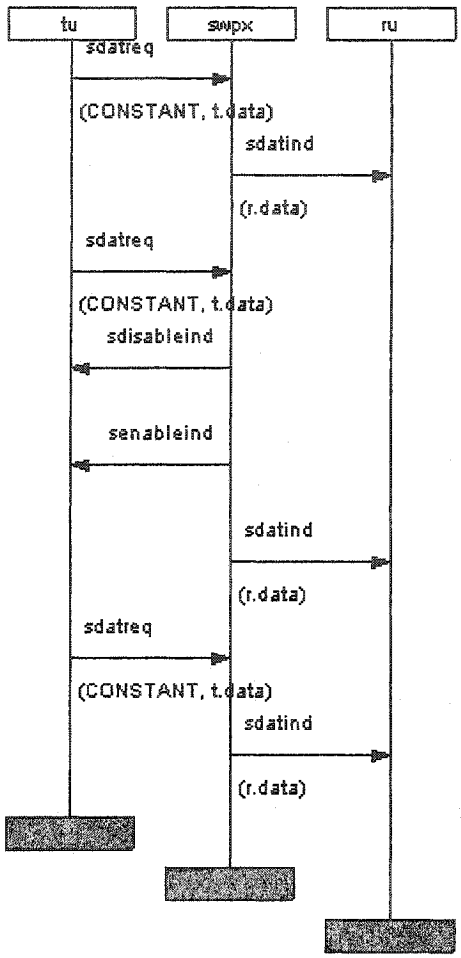


Figure 4.7 A Data Flow Oriented Configuration for Example 2

Since the starting point of our approach is based on a specification, which combines the possible values of control variables with signal names, the signal names have to be changed back manually after the test case generation process, to be consistent with the original specification.

The values of the control variables cannot be simply stripped away from the signal names, because they have to be used as test data for the control variables in the tests. The test data selection for variables, which are not control variables, should be decided by the test designer.

4.6 An Example

Let us consider the Example 2, whose input ".les" file is given in Appendix A.1 and detailed labeled event structure is given in Figure 3.1.

The internal data structure of this example consists of (1) a set of actions, (2) a set of events, (3) a set of component names, (4) a set of messages, (5) a set of start points, (6) a set of cutoff points with the information of their corresponding global state, (7) a set of du-pairs, (8) a set of configurations for control flow oriented testing and (9) a set of configurations for data flow oriented testing.

After the functionality of Phase 1 of the prototype is applied, the first six fields of the internal data structure are filled up with the information of the labeled event structure. Part of the internal data structures is given in Table 4.6.

For example, event "m2" has "m1" as its predecessor, "m4, m6, m8, r1" as its successors and "m!r.mdatindDT(data)" as its action. Action "m!r.mdatindDT(data)" has "m" as its component name, "r" as its remote component name, "mdatindDT" as message and "m.data" as the parameter. As component/remote component name, "m" and "r" belong to the set of component names of the labeled event structure; as a message, "mdatindDT" belongs to the set of messages; as a parameter, "m.data" is added in the list of *c*-uses of the action.

In each EFSM of the system of asynchronously communicating EFSMs, there is one start point. In Example 2, "t0" is the start point for EFSM "t"; "m0" is the start point of EFSM "m"; and "r0" is the start point of EFSM "r".

Cutoff point/global state pairs are also obtained in Phase 1. Each cutoff point of the labeled event structure has one corresponding global state. In Example 2, cutoff point "t9, m9, r3" has "t0, m0, r0" as the corresponding global state; cutoff point "t8, m15, r9" has "t2, m2, r3" as the corresponding global state; cutoff point "t11, m18, r6" has "t6, m9, r3" as the corresponding global state; cutoff point "t14, m21, r6" has "t5, m5, r3" as

the corresponding global state. Theoretically, different cutoff points may have the same corresponding global state in a labeled event structure.

After the functionality of Phase 2 of the prototype is applied, the last three fields of the internal data structure are filled up for the labeled event structure. The set of du-pairs generated for Example 2 is listed in Table 3.7 and Table 3.8.

For example, du-pair (14) in Table 3.7 is generated for timer identifier *timer*1, which is defined at event *r*12 and *c*-used at event *r*15, through the def-clear path *r*12->*r*14->*r*15. Du-pair (N2) in Table 3.8 is generated for variable "r.*j*", which is defined at event *r*20-2 and *c*-used at appended event (*r*20-1), through the def clear path *r*20-2->*r*21->*r*22->*r*9->(*r*7)->(*r*8)->(*r*11)->(*r*13)->(*r*12)->(*r*14)->(*r*15)->(*r*16)->(*r*20-1).

In the procedure of configuration generation for control flow oriented testing, firstly, the prototype generates a set of configurations, each according to one cutoff point of the labeled event structure. In Example 2, four configurations are generated according to the four cutoff points. Then, after the events restriction and inclusions checking of each configuration, some configurations are removed from the set of configurations. The final set of configurations for control flow oriented testing for Example 2 obtained by the prototype is listed below, which satisfies the all-nodes test selection criterion:

1) t1, r2, t3, t5, r5, t11, t12, t14

2) t1, r2, t3, t5, t8, r8

In the procedure of configuration generation for data flow oriented testing, firstly, the prototype generates a set of configurations, each according to one du-pair of the labeled event structure. In Example 2, twenty-one configurations are generated according to the nineteen existing du-pairs and two new du-pairs constructed by the sub-structure appending. Then, after the events restriction and inclusions checking of each configuration, some configurations are removed from the set of configurations. The final set of configurations for data flow oriented testing for Example 2 obtained by the prototype is listed below, which satisfies the all-uses test selection criterion:

1) t1, r2, t3, t5, t11, t12

2) t1, r2, t3, r5

3) t1, r2, t3, t5, t8, r8, (t3), (r8)

After the functionality of phase 3 of the prototype is applied, the textual MSCs of above configurations are obtained. The textual MSCs of this example with their graphic format in Telelogic Tau are given in Appendix A.2 and A.3.

4.7 Another Example

Let us consider another example -- Example 3, whose input ".les" file is given in Appendix B.1 and detailed labeled event structure is given in Figure 4.8.

The complete prefix of the labeled event structure in Figure 4.8 was constructed following the algorithm in [6] from a system of three asynchronously communicating EFSMs. Each of the components add1, add2, and add3 takes two numbers (addend1 and addend2) as input, adds them up, and outputs the sum. To avoid overflowing the recipient's input queue, a sender can carry out a new addition and output the sum only after receiving a confirmation from the recipient. The components add1 and add2 run in parallel. Their output is added together by component add3.

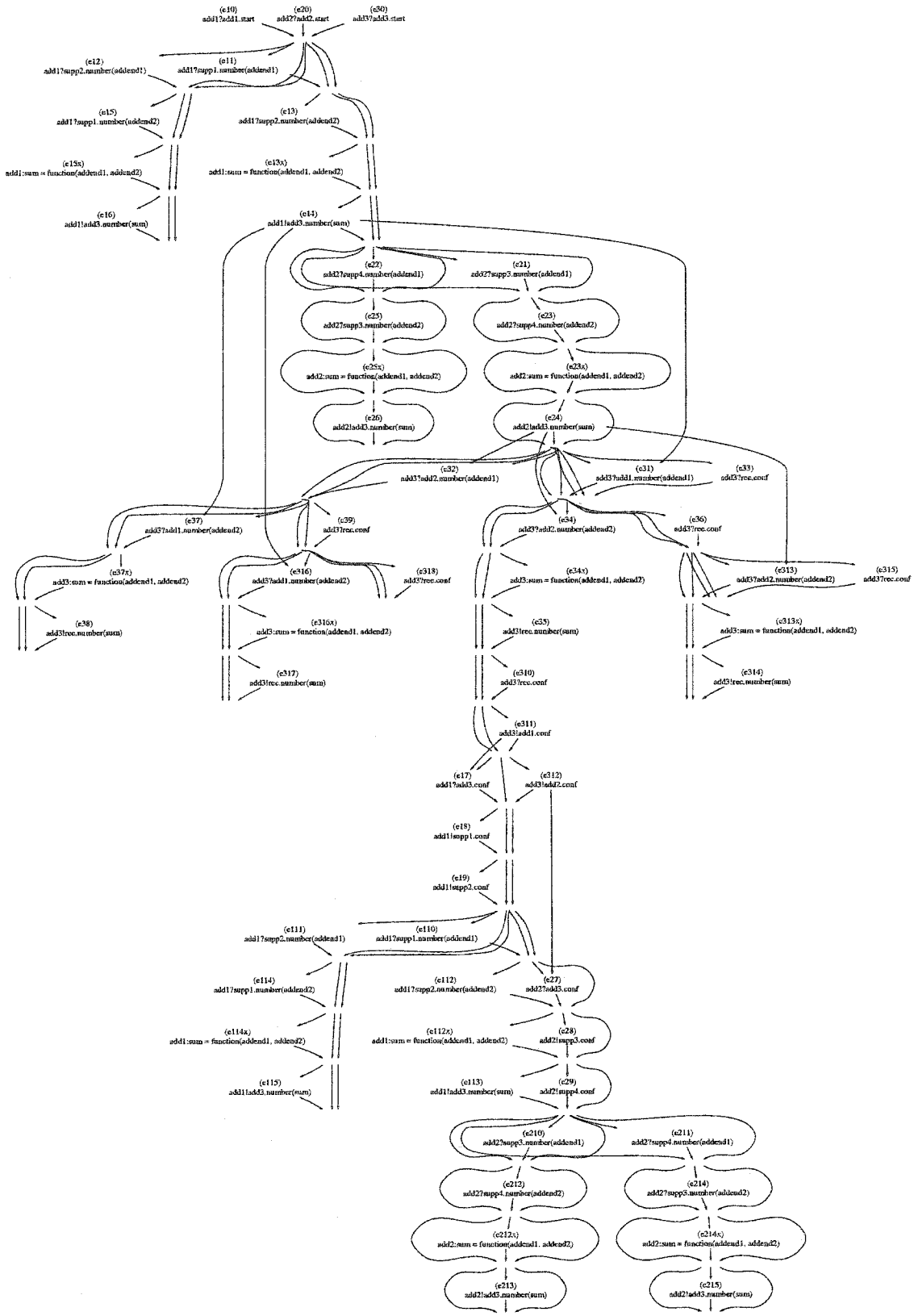


Figure 4.8 Detailed Labeled Event Structure of Example 3

After the functionality of Phase 1 of the prototype is applied, the action set, the event set, the component set, the message set, the start point set and the set of cutoff point/global state pairs of the internal data structure are filled up with the information of the labeled event structure. Part of the internal data structure is given below in Table 4.7.

Table 4.7 Part of Internal Data Structure of Example 3

- **Actions**

Action	loc	mode	rem	msg	para
add1?supp2.number(addend1)	add1	?	supp2	number	add1.addend1
add1:sum:=function (addend1, addend2)	add1	a	add1	∅	add1.addend1, add1.addend2
add1!add3.number(sum)	add1	!	add3	number	add1.sum
add3?add2.number(addend2)	add3	?	add2	number	add3.addend2
add3!rec.conf	add3	!	rec	conf	∅
add3!rec.number(sum)	add3	!	rec	number	add3.sum

- **Events**

Event Id	Action	predecessors	successors
e14	add1!add3.number(sum)	e13x	e17, e31, e37, e316
e115	add1!add3.number(sum)	e114x	∅
e20	add2?add2.start	∅	e22, e21
e27	add2?add3.conf	e24, e312	e28
e37	add3?add1.number(addend2)	e32, e14	e37x
e312	add3!add1.conf	e311	e27

- **Start points**

Start points	e10, e20, e30

- **Cutoff points**

Cutoff point	Corresponding global state
e16, e20, e30	e14, e20, e30
e14, e26, e30	e14, e24, e30
e14, e24, e33	e14, e24, e30
e14, e24, e38	e14, e24, e35
e14, e24, e314	e14, e24, e35
e14, e24, e315	e14, e24, e36
e14, e24, e317	e14, e24, e35
e14, e24, e318	e14, e24, e39
e115, e24, e312	e113, e24, e312
e113, e213, e312	e14, e24, e30
e113, e215, e312	e14, e24, e30

For example, event "e14" has "e13x" as its predecessor, "e17, e31, e37, e316" as its successors and "add1!add3.number(sum)" as its action. Action "add1!add3.number (sum)" has "add1" as its component name, "add3" as its remote component name, "number" as message and "add1.sum" as the parameter. As component/remote component name, "add1" and "add3" belong to the set of component names of the labeled event structure; as a message, "number" belongs to the set of messages; as a parameter, "add1.sum" is added in the list of definition field of the action.

In each EFSM of the system of asynchronously communicating EFSMs, there is one start point. In Example 3, "e10" is the start point for EFSM "add1"; "e20" is the start point of EFSM "add2"; and "e30" is the start point of EFSM "add3".

Cutoff point/global state pairs are also obtained in Phase 1. Each cutoff point of the labeled event structure has one corresponding global state. For example, in Example 3, cutoff point "e16, e20, e30" has "e14, e20, e30" as the corresponding global state; cutoff point "e113, e213, e312" has "e14, e24, e30" as the corresponding global state. Different cutoff points may have the same corresponding global state in a labeled event structure. For example, in Example 3, cutoff point "e14, e24, e314" and "e14, e24, e317" have the same corresponding global state, which is "e14, e24, e35".

After the functionality of Phase 2 of the prototype is applied, the data association set, the du-pair set and the configuration set of the internal data structure are filled up for the labeled event structure. The data dependencies of events of Example 3 are given in Table 4.8, and the set of du-pairs generated for Example 3 is listed in Table 4.9.

Table 4.8 Data Dependencies of Events of Example 3

Event Id	action (act_i)	act_i.defs	act_i.uses
e10	add1?add1.start		
e11	add1?supp1.number(addend1)	add1.addend1	
e12	add1?supp2.number(addend1)	add1.addend1	
e13	add1?supp2.number(addend2)	add1.addend2	
e13x	add1:sum:=function(addend1, addend2)	add1.sum	add1.addend1, add1.addend2
e14	add1!add3.number(sum)		add1.sum
e15	add1?supp1.number(addend2)	add1.addend2	
e15x	add1:sum:=function(addend1,	add1.sum	add1.addend1,

	addend2)		add1.addend2
e16	add1!add3.number(sum)		add1.sum
e17	add1?add3.conf		
e18	add1!supp1.conf		
e19	add1!supp2.conf		
e110	add1?supp1.number(addend1)	add1.addend1	
e111	add1?supp2.number(addend1)	add1.addend1	
e112	add1?supp2.number(addend2)	add1.addend2	
e112x	add1:sum:=function(addend1, addend2)	add1.sum	add1.addend1, add1.addend2
e113	add1!add3.number(sum)		add1.sum
e114	add1?supp1.number(addend2)	add1.addend2	
e114x	add1:sum:=function(addend1, addend2)	add1.sum	add1.addend1, add1.addend2
e115	add1!add3.number(sum)		add1.sum
e20	add2?add2.start		
e21	add2?supp3.number(addend1)	add2.addend1	
e22	add2?supp4.number(addend1)	add2.addend1	
e23	add2?supp4.number(addend2)	add2.addend2	
e23x	add2:sum:=function(addend1, addend2)	add2.sum	add2.addend1, add2.addend2
e24	add2!add3.number(sum)		add2.sum
e25	add2?supp3.number(addend2)	add2.addend2	
e25x	add2:sum:=function(addend1, addend2)	add2.sum	add2.addend1, add2.addend2
e26	add2!add3.number(sum)		add2.sum
e27	add2?add3.conf		
e28	add2!supp3.conf		
e29	add2!supp4.conf		
e210	add2?supp3.number(addend1)	add2.addend1	
e211	add2?supp4.number(addend1)	add2.addend1	
e212	add2?supp4.number(addend2)	add2.addend2	
e212x	add2:sum:=function(addend1, addend2)	add2.sum	add2.addend1, add2.addend2
e213	add2!add3.number(sum)		add2.sum
e214	add2?supp3.number(addend2)	add2.addend2	
e214x	add2:sum:=function(addend1, addend2)	add2.sum	add2.addend1, add2.addend2
e215	add2!add3.number(sum)		add2.sum
e30	add3?add3.start		
e31	add3?add1.number(addend1)	add3.addend1	
e32	add3?add2.number(addend1)	add3.addend1	
e33	add3?rec.conf		
e34	add3?add2.number(addend2)	add3.addend2	
e34x	add3:sum:=function(addend1,	add3.sum	add3.addend1,

	addend2)		add3.addend2
e35	add3!rec.number(sum)		add3.sum
e36	add3?rec.conf		
e37	add3?add1.number(addend2)	add3.addend2	
e37x	add3:sum:=function(addend1, addend2)	add3.sum	add3.addend1, add3.addend2
e38	add3!rec.number(sum)		add3.sum
e39	add3?rec.conf		
e310	add3?rec.conf		
e311	add3!add1.conf		
e312	add3!add2.conf		
e313	add3?add2.number(addend2)	add3.addend2	
e313x	add3:sum:=function(addend1, addend2)	add3.sum	add3.addend1, add3.addend2
e314	add3!rec.number(sum)		add3.sum
e315	add3?rec.conf		
e316	add3?add1.number(addend2)	add3.addend2	
e316x	add3:sum:=function(addend1, addend2)	add3.sum	add3.addend1, add3.addend2
e317	add3!rec.number(sum)		add3.sum
e318	add3?rec.conf		

Table 4.9 du-pairs of Example 3

No.	Com	variable	def event	cuse event	def-clear path
1	add1	add1.addend1	e11	e13x	e11->e13->e13x
2	add1	add1.addend2	e13	e13x	e13->e13x
3	add1	add1.sum	e13x	e14	e13x->e14
4	add1	add1.addend1	e12	e15x	e12->e15->e15x
5	add1	add1.addend2	e15	e15x	e15->e15x
6	add1	add1.sum	e15x	e16	e15x->e16
7	add1	add1.addend1	e110	e112x	e110->e112->e112x
8	add1	add1.addend2	e112	e112x	e112->e112x
9	add1	add1.sum	e112x	e113	e112x->e113
10	add1	add1.addend1	e111	e114x	e111->e114->e114x
11	add1	add1.addend2	e114	e114x	e114->e114x
12	add1	add1.sum	e114x	e115	e114x->e115
13	add2	add2.addend1	e21	e23x	e21->e23->e23x
14	add2	add2.addend2	e23	e23x	e23->e23x
15	add2	add2.sum	e23x	e24	e23x->e24
16	add2	add2.addend1	e22	e25x	e22->e25->e25x
17	add2	add2.addend2	e25	e25x	e25->e25x
18	add2	add2.sum	e25x	e26	e25x->e26
19	add2	add2.addend1	e210	e212x	e210->e212->e212x
20	add2	add2.addend2	e212	e212x	e212->e212x

21	add2	add2.sum	e212x	e213	e212x->e213
22	add2	add2.addend1	e211	e214x	e211->e214->e214x
23	add2	add2.addend2	e214	e214x	e214->e214x
24	add2	add2.sum	e214x	e215	e214x->e215
25	add3	add3.addend1	e31	e34x	e31->e34->e34x
26	add3	add3.addend2	e34	e34x	e34->e34x
27	add3	add3.sum	e34x	e35	e34x->e35
28	add3	add3.addend1	e32	e37x	e32->e37->e37x
29	add3	add3.addend2	e37	e37x	e37->e37x
30	add3	add3.sum	e37x	e38	e37x->e38
31	add3	add3.addend1	e31	e313x	e31->e36->e313->e313x
32	add3	add3.addend2	e313	e313x	e313->e313x
33	add3	add3.sum	e313x	e314	e313x->e314
34	add3	add3.addend1	e32	e316x	e32->e39->e316->e316x
35	add3	add3.addend2	e316	e316x	e316->e316x
36	add3	add3.sum	e316x	e317	e316x->e317

For example, du-pair (31) in Table 4.9 is generated for variable "add3.addend1" of component "add3". It is defined at event *e*31 and *c*-used at event *e*313x, through the def-clear path *e*31->*e*36->*e*313->*e*313x.

In Example 3, all the definitions in the labeled event structure are being *c*-used, so that there is no need to append sub-structures to create du-pairs.

In the procedure of configuration generation for control flow oriented testing, firstly, the prototype generates a set of configurations, each according to one cutoff point of the labeled event structure. In Example 3, corresponding to the 11 cutoff points of the labeled event structure, 11 configurations are generated according to the 11 cutoff points. After the events restriction and inclusions checking of each configuration, no configuration is removed from the set of configurations, and only the events occurring at PCOs are remained in the final configuration. The final set of configurations for control flow oriented testing for Example 3 obtained by the prototype is listed below, which satisfies the all-nodes test selection criterion:

1) e12, e15

2) e11, e13, e21, e23, e38

3) e11, e13, e21, e23, e39, e317

4) e11, e13, e21, e23, e39, e318

5) e11, e13, e22, e25

6) e11, e13, e21, e23, e35, e310, e18, e19, e111, e114

7) e11, e13, e21, e23, e35, e310, e18, e19, e110, e112, e28, e29, e210, e212

8) e11, e13, e21, e23, e35, e310, e18, e19, e110, e112, e28, e29, e211, e214

9) e11, e13, e21, e23, e33

10) e11, e13, e21, e23, e36, e314

11) e11, e13, e21, e23, e36, e315

In the procedure of configuration generation for data flow oriented testing, firstly, the prototype generates a set of configurations, each according to one du-pair of the labeled event structure. In Example 3, 36 configurations are generated according to the 36 existing du-pairs. Then, after the events restriction and inclusions checking of each configuration, some configurations are removed from the set of configurations. The final set of configurations for data flow oriented testing for Example 3 obtained by the prototype is listed below, which satisfies the all-uses test selection criterion:

1) e12, e15

2) e11, e13, e21, e23, e35, e310, e18, e19, e110, e112

3) e11, e13, e21, e23, e35, e310, e18, e19, e111, e114

4) e22, e25

5) e11, e13, e21, e23, e35, e310, e28, e29, e210, e212

6) e11, e13, e21, e23, e35, e310, e28, e29, e211, e214

7) e11, e13, e21, e23, e38

8) e11, e13, e21, e23, e36, e314

9) e11, e13, e21, e23, e39, e317

After the functionality of Phase 3 of the prototype is applied, the textual MSCs of above configurations are obtained. The textual MSCs of this example with their graphic format in Telelogic Tau are given in Appendix B.2 and B.3.

The source code of the prototype can be found at www.site.uottawa.ca/~ural.

Chapter 5 Conclusions

5.1 Comparison with Related Work

Control and data flow aspects of a distributed system can be identified through the analysis of control and data dependencies that exist not only within processes, but also across process boundaries. Test generation from single EFSM's [2, 3, 5, 19, 20, 21, 23] and also from systems of asynchronously communicating FSM's or EFSM's [6, 7, 8, 34, 35, 36] has been widely studied. The work studied in this thesis is based on these research.

Compared to test generation methods that require an explicit specification of test purposes as input, the methods with implicit test purposes require less manual efforts and guarantee a systematic test coverage. [19, 20] abstract the control dependencies in the EFSM representation of a process as an FSM and apply FSM based test generation methods. [2, 3, 5, 21, 23] identify the data dependencies in the EFSM representation of a process by applying principles of functional program testing [32] or data flow analysis [33]. All methods for test generation from EFSM's face the feasibility problem, i.e. the problem to decide which sequences of transitions are feasible and for which values of the variables are they feasible. Because of enabling conditions, a sequence of transitions may be infeasible for all or some values of the variables. The feasibility problem is undecidable in general. It can be solved, however, in certain restricted cases (if the variables influencing the feasibility take on only finitely many discrete values) [3, 6, 23].

The methods discussed so far consider only a single EFSM and hence their applicability is restricted to sequential systems. On the other hand, some other methods proposed with implicit test purposes for systems of communicating EFSMs suffer from the state-explosion problem. Different approaches to alleviate the state-explosion problem have been proposed. [34] pursues an approach similar to program slicing, pruning the given communicating FSM's to contain only a subset of actions, thus yielding a set of smaller, simplified specifications. [7, 35] aim at diminishing the state

explosion by generating non-interleaving models of the original specification by a reduced reachability analysis approach. TestComposer [36] makes use of a reduced reachability analysis approach and implies as test purposes all transitions in specification. Although it is a step in the right direction, the implied test purposes do not represent the set of functionalities in the given specification due to the fact that only an ordered set of individual transitions is a representation of a specific functionality of the system.

This thesis complements the approach in [6] and implements a method that generates test cases with implicit purpose, exposing control and data dependencies within a complete prefix of the labeled event structure of a system of asynchronously communicating EFSMs.

5.2 Results of this thesis

The approach introduced in this thesis generates tests in MSC from a system of asynchronously communicating EFSMs. It is based on a non-interleaving model, called labeled event structure. This model is intended for the generation of tests through the application of control flow oriented as well as data flow oriented test selection criteria [10, 22, 37] proposed in the literature for software testing. It facilitates the application of test criteria by exposing the intra-process control and/or data dependencies within each process and the inter-process control and/or data dependencies among communicating processes. It enlarges the syntax of EFSMs by supporting of all kinds of actions that can occur in an EFSM: input, output, assignment, set, reset, timeout, procedure call. This model also alleviates the state-explosion problem and ensures the feasibility for all the tests by switching the possible values of control variables to be combined with input/output signals.

The algorithm for generating a labeled event structure from a generalized model of asynchronously communicating EFSMs is given in [6], and the algorithm for generating a set of test cases from the labeled event structure is given in this thesis.

The prototype tool that aims at generating test cases from the model of labeled event structure is implemented and applied to some examples given in the Appendixes.

For control flow oriented aspects, the algorithm achieves 100% all-nodes coverage of the nodes of a complete prefix of the labeled event structure of a system of asynchronously communicating EFSMs. For data flow oriented aspects, the algorithm achieves 100% all-uses coverage within the complete prefix of the labeled event structure.

The applicability of this approach is restricted to systems of asynchronously communicating EFSMs where control variables have only a finite number of discrete values, and the contents of first in first out queues are bounded.

5.3 Future Enhancements

One area for future work is to investigate the solutions for the problems raised by the increasing of number of possible values of control variables.

The second area for future work is to optimize the set of data flow oriented configurations. So far, the set of configurations for data flow oriented testing are du-pair oriented. Although they are not included into each other, because they are not in conflict, to each other, some of them can be merged together. For example, in Example 2, configuration "t1, r2, t3, t5, t11, t12" and "t1, r2, t3, r5" can be merged into one configuration as "t1, r2, t3, t5, r5, t11, t12".

Another area of future enhancement is to merge the two sets of configurations (one set is for control flow oriented testing, the other set is for data flow oriented testing) into one set, and the inclusion checking should be applied to the final set of configurations.

Steps 2 and 3 of Phase 3 of the approach could be automated. The requirements for the automation of steps 2 and 3 are: the mapping information between the original signal names and the new signal names enhanced with values of control variables are known, and the values of control variables are known.

The coverage of the nodes and branches of the original specification by the set of tests generated by our approach can be measured using Telelogic Tau. Such a coverage study provides an excellent feedback for the tester on the quality of the test suite [37].

References

[1] V. Sassone, M. Nielsen, G. Winskel, "Models for concurrency: Towards a classification", Theoretical Computer Science, 170, pp.297-348, 1996

[2] H. Ural and B. Yang, "A test sequence generation method for protocol testing", IEEE Trans. Comm., 39, 4, pp. 514-523, 1991

[3] O. Henniger, A. Ulrich, and H. König, "Transformation of Estelle modules aiming at test case generation", in Proc. of IWPTS'95, Evry, France, 1995

[4] L.A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A Formal Evaluation of Data Flow Path Selection Criteria", IEEE Trans. Software Eng., 15, 11, pp. 1318-1332, 1989

[5] H. Ural, K. Saleh, A. Williams, "Test generation based on control and data dependencies within system specifications in SDL", Computer Communications, 23, pp. 609-627, 2000

[6] O. Henniger, "Test generation from specifications in Estelle and SDL", Brandenburg University of Technology Cottbus, Germany, PhD thesis, 2002, in German

[7] O. Henniger, "On test case generation from asynchronously communicating state machines", in Proc. of IWTCS'97, Cheju Island, South Korea, 1997

[8] O. Henniger, H. Ural, "Test generation based on control and data dependencies within multi-process SDL specifications", in Proc. of SAM 2000, Col de Porte, Grenoble, France, 2000

[9] M.J. Harrold and M.L. Soffa, "Interprocedural data flow testing", in Proc. Of 3rd Symposium on Testing, Analysis, and Verification, Key West, Florida, pp. 158-167, 1989

[10] S. Rapps and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information", IEEE Trans. Software Eng., 11, 4, pp. 367-375, 1985

[11] Information technology – Open Systems Interconnection – Conformance testing methodology and framework. International Standard ISO/IEC 9646, 1991

[12] J.W. Laski and B. Korel, "A data-flow oriented program testing strategy", IEEE Trans. Software Eng., 9, 5, pp. 347-354, 1983

[13] J. Esparza, S. Römer, and W. Vogle, "An improvement of McMillan's unfolding algorithm", In T. Margaria and B. Steffen, editors, Tools and Algorithms for the construction and Analysis of System, pages 87-106, Passau, Germany, 1996. Springer-Verlag

[14] J.-C. Fernandez, C. Jard, T. Jeron, and C. Viho, "Using on-the-fly verification techniques for the generation of test suites", in Proc. of CAV'96, New Brunswich, NJ, USA, pp. 348-359, 1996

[15] J. Grabowski, D. Hogrefe, and R. Nahm, "Test case generation with test purpose specification by MSCs", in Proc. of the 6th SDL Forum, Darmstadt, Germany, 1993

[16] R. Groz and N. Risser, "Eight years of experience in test generation from FDTs using TVEDA", in Proc. of FORTE/PSTV'97, Chapman and Hall, 1997

[17] ObjectGeode, Verilog, Toulouse, France, 1996

[18] G.J. Myers, The art of software testing. New York: John Wiley & Sons, 1979

[19] A. T. Dahbura, K.K. Sabnani, and M.Ü. Uyar, "Formal methods for generating protocol conformance test sequences", Proceedings of the IEEE, 78, 8, 1990, pp. 1317-1325

[20] D.P. Sidhu and T.K. Leung, "Formal methods for protocol testing: A detailed study", IEEE Trans. Software Eng., 15, 4, pp. 413-426, 1989

[21] B. Sarikaya, G.v. Bochmann, and E. Cerny, "A test design methodology for protocol testing", IEEE Trans. Software Eng., 13, 5, pp. 518-531, 1987

[22] P. G. Frankl, and E. J. Weyuker, "An Applicable Family of Data Flow Testing Criteria", IEEE Trans. Software Eng., 14, 10, pp. 1483-1498, 1988

[23] W. Chun and P.D. Amer, "Test case generation for protocols specified in Estelle", in Proc. Of FORTE'90, Madrid, Spain, pp. 197-210, 1990

[24] Information processing systems – Open Systems Interconnection – Estelle: A formal description technique based on an extended state transition model. International Standard ISO 9074, 1989

[25] Specification and Description Language (SDL), ITU-T Recommendation Z.100, 1992

[26] B. Baumgarten and A. Giessler. OSI conformance testing methodology and TTCN. North-Holland, 1994

[27] Telelogic Tau Help file

[28] Information processing systems – Open Systems Interconnection – LOTOS: A formal description technique based on the temporal ordering of observational behavior. International Standard ISO 8807, 1989

[29] Telelogic Tau 3.4, Telelogic AB, Malmö, Sweden, 1998

[30] ITU Recommendation Z.120, Message Sequence Charts (MSC).

[31] B. Beizer, Software Testing Techniques, Second Edition, Van Nostrand Reinhold Inc. 1990

[32] W.E. Howden, Functional program testing and analysis, McGraw Hill, New York, 1987

[33] L.D. Fosdick and L.J. Osterweil, "Data flow analysis in software reliability", ACM Computing Surveys, 8, 3, pp. 305-330, 1976

[34] D. Lee, K.K. Sabnani, D.M. Kristol, S. Paul, "Conformance testing of protocols specified as communicating FSMs", in Proc. Of IEEE INFOCOM'93, San Francisco, CA, USA, 1993

[35] N. Arakawa and T. Soneoka, "A test case generation method for concurrent programs", in Proc. Of IWPTS'91, Leidschendam, The Netherlands, 1991

[36] A. Kerbrat, T. Jeron, and R. Groz, "Automated test generation from SDL specifications", in Proc. Of SDL Forum'99, Montreal, Canada, pp. 135-151, 1999

[37] R.L. Probert, H. Ural, A.W. Williams, "Rapid generation of functional tests using MSCs, SDL and TTCN", Computer Communications, 24, pp. 374-393, 2001

Appendix A An EFSM Based Example (swpx.les)

A.1 Labeled Event Structure File

```
label(et0)  = t?t.start;
label(et1)  = t?tu.sdatreq(constant,data);
label(et2)  = t!m.mdatreqDT(data);
label(et3)  = t?tu.sdatreq(constant,data);
label(et4)  = t!m.mdatreqDT(data);
label(et5)  = t!tu.sdisableind;
label(et6)  = t?t.nil;
label(et7)  = t?m.mdatindAK;
label(et8)  = t!tu.senableind;
label(et9)  = t?m.mdatindAK;
label(et10) = t?m.mdatindAK;
label(et11) = t!tu.senableind;
label(et12) = t?tu.sdatreq(constant,data);
label(et13) = t!m.mdatreqDT(data);
label(et14) = t!tu.sdisableind;

label(em0)  = m?m.start;
label(em1)  = m?t.mdatreqDT(data);
label(em2)  = m!r.mdatindDT(data);
label(em4)  = m?t.mdatreqDT(data);
label(em5)  = m!r.mdatindDT(data);
label(em6)  = m?r.mdatreqAK(constant);
label(em7)  = m!t.mdatindAK;
label(em8)  = m?r.mdatreqAK(constant);
label(em9)  = m!t.mdatindAK;
label(em11) = m?r.mdatreqAK(constant);
label(em12) = m!t.mdatindAK;
label(em14) = m?t.mdatreqDT(data);
label(em15) = m!r.mdatindDT(data);
label(em17) = m?r.mdatreqAK(constant);
label(em18) = m!t.mdatindAK;
label(em19) = m?m.nil;
label(em20) = m?t.mdatreqDT(data);
label(em21) = m!r.mdatindDT(data);

label(er0)  = r?r.start;
label(er1)  = r?m.mdatindDT(data);
label(er2)  = r!ru.sdatind(data);
label(er3)  = r!m.mdatreqAK(constant);
```

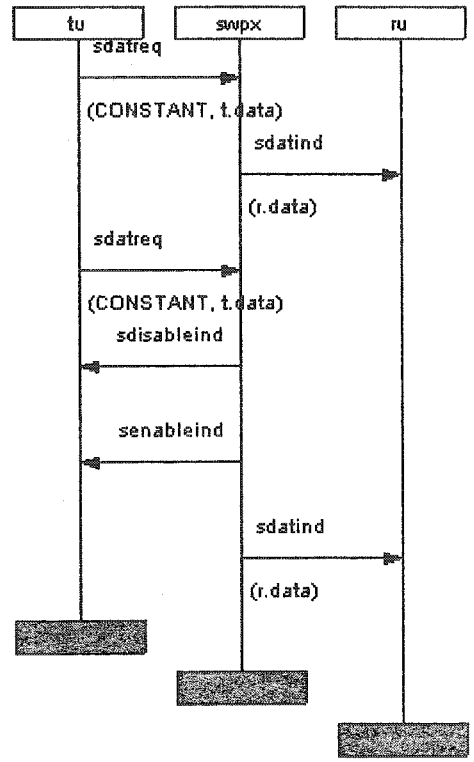
```
label(er4)  = r?m.mdatindDT(data);
label(er5)  = r!ru.sdatind(data);
label(er6)  = r!m.mdatreqAK(constant);
label(er7)  = r?m.mdatindDT(data);
label(er8)  = r!ru.sdatind(data);
label(er9)  = r!m.mdatreqAK(constant);
label(er10) = r:a:=constant;
label(er11) = r:b:=constant;
label(er12) = r:set(constant,timer1);
label(er13) = r:set(constant,timer2);
label(er14) = r:reset(timer2);
label(er15) = r?r.timer1;
label(er16) = r:a:=function(a,b);
label(er18) = r:procedure(pc2(x,y,j;a)) {x:=function(a); y:=function(x); j:=
constant;};
label(er20) = r:procedure(pc4(x,j;y){x:=function(j,y);j:=constant;};
label(er21) = r:set(constant,timer3);
label(er22) = r?r.timer3;

root = {et0, em0, er0};

et0 < et1 < et2;
et2 < et3 < et4 < et5;
et2 < et6 < et9;
et5 < et10 < et11 < et12 < et13 < et14;
et5 < et7 < et8;

et2 < em1;
et4 < em4;
et4 < em14;
et13 < em20;
em7 < et7;
em9 < et9;
em12 < et10;

em0 < em1 < em2;
em2 < em4 < em5 < em11 < em12;
em2 < em6 < em7 < em14 < em15;
em2 < em8 < em9;
em12 < em17 < em18;
em12 < em19 < em20 < em21;

em2 < er1;
em5 < er4;
em15 < er7;
er3 < em6;
```

```
er3  < em8;
er3  < em11;
er6  < em17;

er0  < er1  < er2  < er10 < er18 < er3;
er3  < er4  < er5  < er6;
er3  < er7  < er8  < er11 < er13 < er12 < er14 < er15 < er16 < er20 < er21 < er22
< er9;
er12 < er15;
er13 < er14;
er21 < er22;

et3  # et6;
et3  # em8;
et6  # em6;
et12 # em17;
em4  # em6;
em6  # em8;
em17 # em19;

{et9, em9, er3} -> {et0, em0, er0};
{et8, em15, er9} -> {et2, em2, er3};
{et11, em18, er6} -> {et6, em9, er3};
{et14, em21, er6} -> {et5, em5, er3};
```

A.2 Configurations for Control Flow Oriented Testing

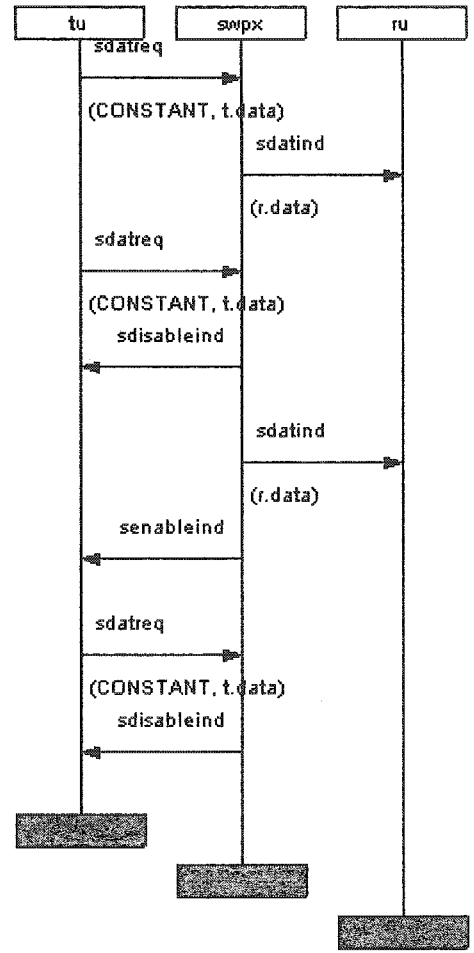
The first configuration for control flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
tu: instancehead;
swpx: instancehead;
ru: instancehead;
tu: out sdatreq,1(CONSTANT, t.data) to swpx;
swpx: in sdatreq,1(CONSTANT, t.data) from tu;
swpx: out sdatind,2(r.data) to ru;
ru: in sdatind,2(r.data) from swpx;
tu: out sdatreq,3(CONSTANT, t.data) to swpx;
swpx: in sdatreq,3(CONSTANT, t.data) from tu;
swpx: out sdisableind,4 to tu;
tu: in sdisableind,4 from swpx;
swpx: out senableind,5 to tu;
tu: in senableind,5 from swpx;
swpx: out sdatind,6(r.data) to ru;
ru: in sdatind,6(r.data) from swpx;
tu: endinstance;
swpx: endinstance;
ru: endinstance;
endmsc;
	

The second configuration for control flow oriented testing:

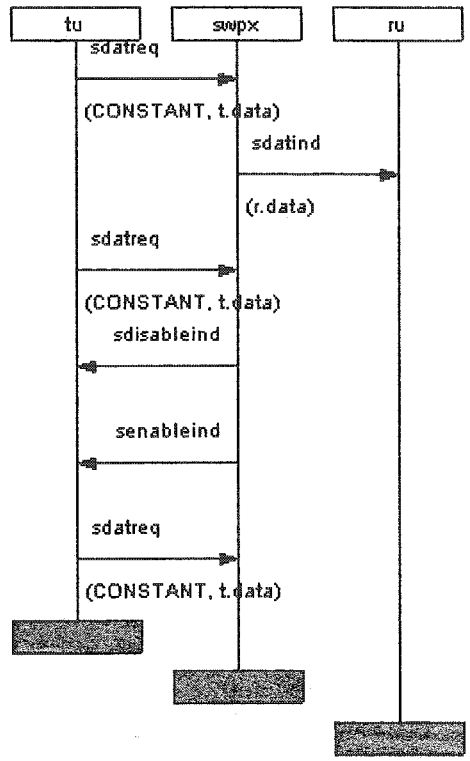
Textual format MSC	Graphical format MSC

msc configuration;
tu: instancehead;
swpx: instancehead;
ru: instancehead;
tu: out sdatreq,1(CONSTANT, t.data) to swpx;
swpx: in sdatreq,1(CONSTANT, t.data) from tu;
swpx: out sdatind,2(r.data) to ru;
ru: in sdatind,2(r.data) from swpx;
tu: out sdatreq,3(CONSTANT, t.data) to swpx;
swpx: in sdatreq,3(CONSTANT, t.data) from tu;
swpx: out sdisableind,4 to tu;
tu: in sdisableind,4 from swpx;
swpx: out sdatind,5(r.data) to ru;
ru: in sdatind,5(r.data) from swpx;
swpx: out senableind,6 to tu;
tu: in senableind,6 from swpx;
tu: out sdatreq,7(CONSTANT, t.data) to swpx;
swpx: in sdatreq,7(CONSTANT, t.data) from tu;
swpx: out sdisableind,8 to tu;
tu: in sdisableind,8 from swpx;
tu: endinstance;
swpx: endinstance;
ru: endinstance;
endmsc;
	

A.3 Configurations for Data Flow Oriented Testing

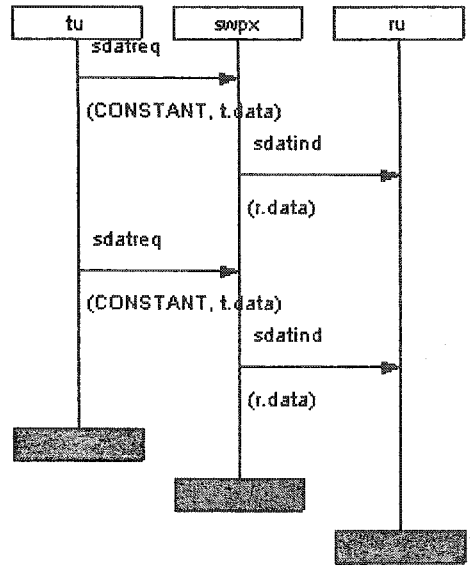
The first configuration for data flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
tu: instancehead;
swpx: instancehead;
ru: instancehead;
tu: out sdatreq,1(CONSTANT, t.data) to swpx;
swpx: in sdatreq,1(CONSTANT, t.data) from tu;
swpx: out sdatind,2(r.data) to ru;
ru: in sdatind,2(r.data) from swpx;
tu: out sdatreq,3(CONSTANT, t.data) to swpx;
swpx: in sdatreq,3(CONSTANT, t.data) from tu;
swpx: out sdisableind,4 to tu;
tu: in sdisableind,4 from swpx;
swpx: out senableind,5 to tu;
tu: in senableind,5 from swpx;
tu: out sdatreq,6(CONSTANT, t.data) to swpx;
swpx: in sdatreq,6(CONSTANT, t.data) from tu;
tu: endinstance;
swpx: endinstance;
ru: endinstance;
endmsc;
	

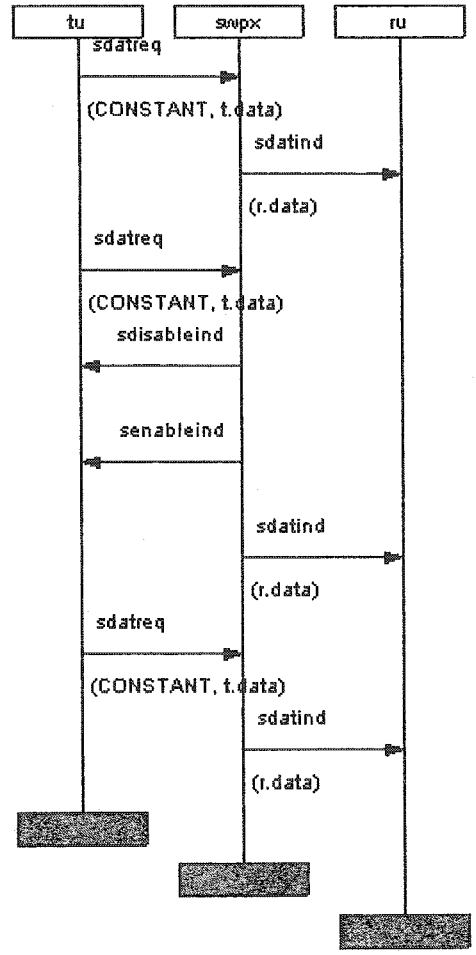
The second configuration for data flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
tu: instancehead;
swpx: instancehead;
ru: instancehead;
tu: out sdatreq,1(CONSTANT, t.data) to swpx;
swpx: in sdatreq,1(CONSTANT, t.data) from tu;
swpx: out sdatind,2(r.data) to ru;
ru: in sdatind,2(r.data) from swpx;
tu: out sdatreq,3(CONSTANT, t.data) to swpx;
swpx: in sdatreq,3(CONSTANT, t.data) from tu;
swpx: out sdatind,4(r.data) to ru;
ru: in sdatind,4(r.data) from swpx;
tu: endinstance;
swpx: endinstance;
ru: endinstance;
endmsc;
	

The third configuration for data flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
tu: instancehead;
swpx: instancehead;
ru: instancehead;
tu: out sdatreq,1(CONSTANT, t.data) to swpx;
swpx: in sdatreq,1(CONSTANT, t.data) from tu;
swpx: out sdatind,2(r.data) to ru;
ru: in sdatind,2(r.data) from swpx;
tu: out sdatreq,3(CONSTANT, t.data) to swpx;
swpx: in sdatreq,3(CONSTANT, t.data) from tu;
swpx: out sdisableind,4 to tu;
tu: in sdisableind,4 from swpx;
swpx: out senableind,5 to tu;
tu: in senableind,5 from swpx;
swpx: out sdatind,6(r.data) to ru;
ru: in sdatind,6(r.data) from swpx;
tu: out sdatreq,7(CONSTANT, t.data) to swpx;
swpx: in sdatreq,7(CONSTANT, t.data) from tu;
swpx: out sdatind,8(r.data) to ru;
ru: in sdatind,8(r.data) from swpx;
tu: endinstance;
swpx: endinstance;
ru: endinstance;
endmsc;
	

Appendix B An EFSM Based Example (adderx.les)

B.1 Labeled Event Structure File

```
label(e10) = add1?add1.start;
label(e11) = add1?supp1.number(addend1);
label(e12) = add1?supp2.number(addend1);
label(e13) = add1?supp2.number(addend2);
label(e13x) = add1:sum:=function(addend1,addend2);
label(e14) = add1!add3.number(sum);
label(e15) = add1?supp1.number(addend2);
label(e15x) = add1:sum:=function(addend1,addend2);
label(e16) = add1!add3.number(sum);
label(e17) = add1?add3.conf;
label(e18) = add1!supp1.conf;
label(e19) = add1!supp2.conf;
label(e110) = add1?supp1.number(addend1);
label(e111) = add1?supp2.number(addend1);
label(e112) = add1?supp2.number(addend2);
label(e112x)= add1:sum:=function(addend1,addend2);
label(e113) = add1!add3.number(sum);
label(e114) = add1?supp1.number(addend2);
label(e114x)= add1:sum:=function(addend1,addend2);
label(e115) = add1!add3.number(sum);

label(e20) = add2?add2.start;
label(e21) = add2?supp3.number(addend1);
label(e22) = add2?supp4.number(addend1);
label(e23) = add2?supp4.number(addend2);
label(e23x) = add2:sum:=function(addend1,addend2);
label(e24) = add2!add3.number(sum);
label(e25) = add2?supp3.number(addend2);
label(e25x) = add2:sum:=function(addend1,addend2);
label(e26) = add2!add3.number(sum);
label(e27) = add2?add3.conf;
label(e28) = add2!supp3.conf;
label(e29) = add2!supp4.conf;
label(e210) = add2?supp3.number(addend1);
label(e211) = add2?supp4.number(addend1);
label(e212) = add2?supp4.number(addend2);
label(e212x)= add2:sum:=function(addend1,addend2);
label(e213) = add2!add3.number(sum);
label(e214) = add2?supp3.number(addend2);
label(e214x)= add2:sum:=function(addend1,addend2);
```

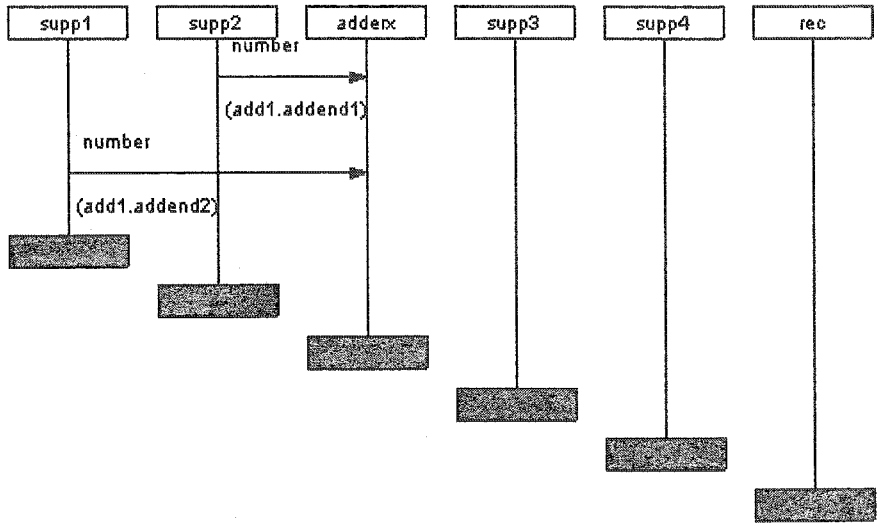
```
label(e215) = add2!add3.number(sum);

label(e30)  = add3?add3.start;
label(e31)  = add3?add1.number(addend1);
label(e32)  = add3?add2.number(addend1);
label(e33)  = add3?rec.conf;
label(e34)  = add3?add2.number(addend2);
label(e34x) = add3:sum:=function(addend1,addend2);
label(e35)  = add3!rec.number(sum);
label(e36)  = add3?rec.conf;
label(e37)  = add3?add1.number(addend2);
label(e37x) = add3:sum:=function(addend1,addend2);
label(e38)  = add3!rec.number(sum);
label(e39)  = add3?rec.conf;
label(e310) = add3?rec.conf;
label(e311) = add3!add1.conf;
label(e312) = add3!add2.conf;
label(e313) = add3?add2.number(addend2);
label(e313x)= add3:sum:=function(addend1,addend2);
label(e314) = add3!rec.number(sum);
label(e315) = add3?rec.conf;
label(e316) = add3?add1.number(addend2);
label(e316x)= add3:sum:=function(addend1,addend2);
label(e317) = add3!rec.number(sum);
label(e318) = add3?rec.conf;

root = {e10, e20, e30};

e10 < e11 < e13 < e13x < e14 < e17 < e18 < e19 < e110 < e112 < e112x <
e113;
e10 < e12 < e15 < e15x < e16;
e19 < e111 < e114 < e114x< e115;

e14 < e31;
e14 < e37;
e14 < e316;

e20 < e21 < e23 < e23x < e24 < e27 < e28 < e29 < e210 < e212 < e212x <
e213;
e20 < e22 < e25 < e25x < e26;
e29 < e211 < e214 < e214x< e215;

e24 < e32;
e24 < e34;
e24 < e313;
```

```
e30 < e31 < e34 < e34x < e35 < e310 < e311 < e312;
e31 < e36 < e313 < e313x< e314;
e36 < e315;
e30 < e32 < e37 < e37x < e38;
e32 < e39 < e316 < e316x< e317;
e39 < e318;
e30 < e33;

e311 < e17;
e312 < e27;

e11 # e12;
e12 # e21;
e12 # e22;
e12 # e31;
e12 # e32;
e12 # e33;
e110 # e111;
e111 # e27;
e21 # e22;
e22 # e31;
e22 # e33;
e210 # e211;
e31 # e32;
e31 # e33;
e32 # e33;
e34 # e36;
e37 # e39;
e313 # e315;
e316 # e318;

{e16, e20, e30} -> {e14, e20, e30};
{e14, e26, e30} -> {e14, e24, e30};
{e14, e24, e33} -> {e14, e24, e30};
{e14, e24, e38} -> {e14, e24, e35};
{e14, e24, e314} -> {e14, e24, e35};
{e14, e24, e315} -> {e14, e24, e36};
{e14, e24, e317} -> {e14, e24, e35};
{e14, e24, e318} -> {e14, e24, e39};
{e115, e24, e312} -> {e113, e24, e312};
{e113, e213, e312} -> {e14, e24, e30};
{e113, e215, e312} -> {e14, e24, e30};
```

B.2 Configurations for Control Flow Oriented Testing

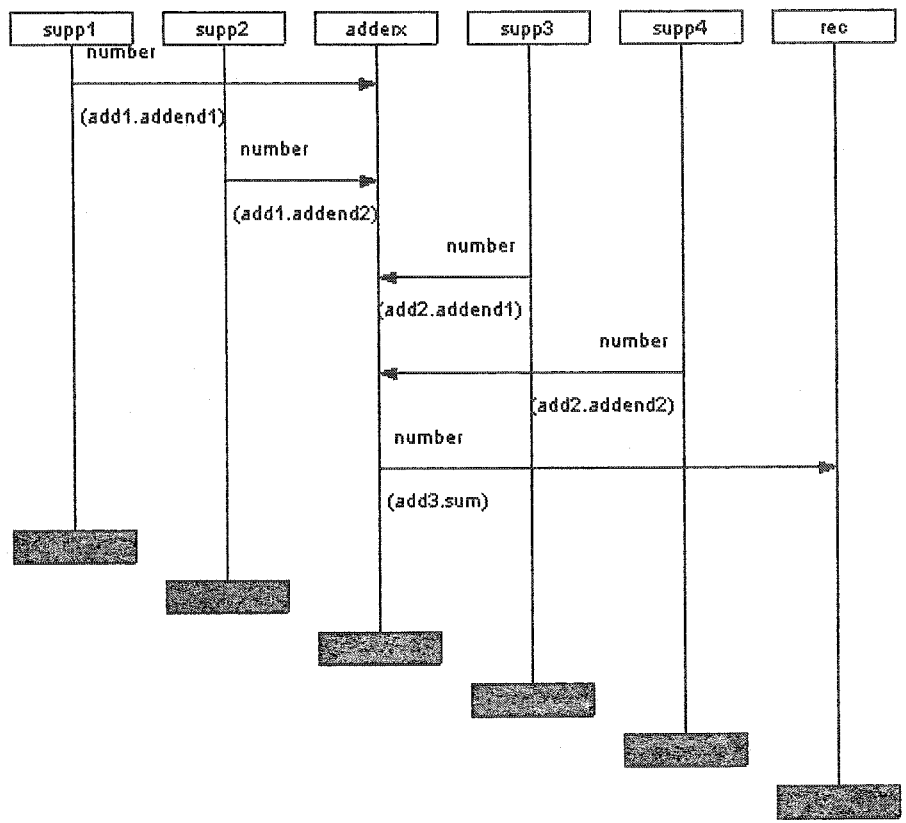
The first configuration for control flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp2: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp2;
supp1: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp1;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
	

The second configuration for control flow oriented testing:

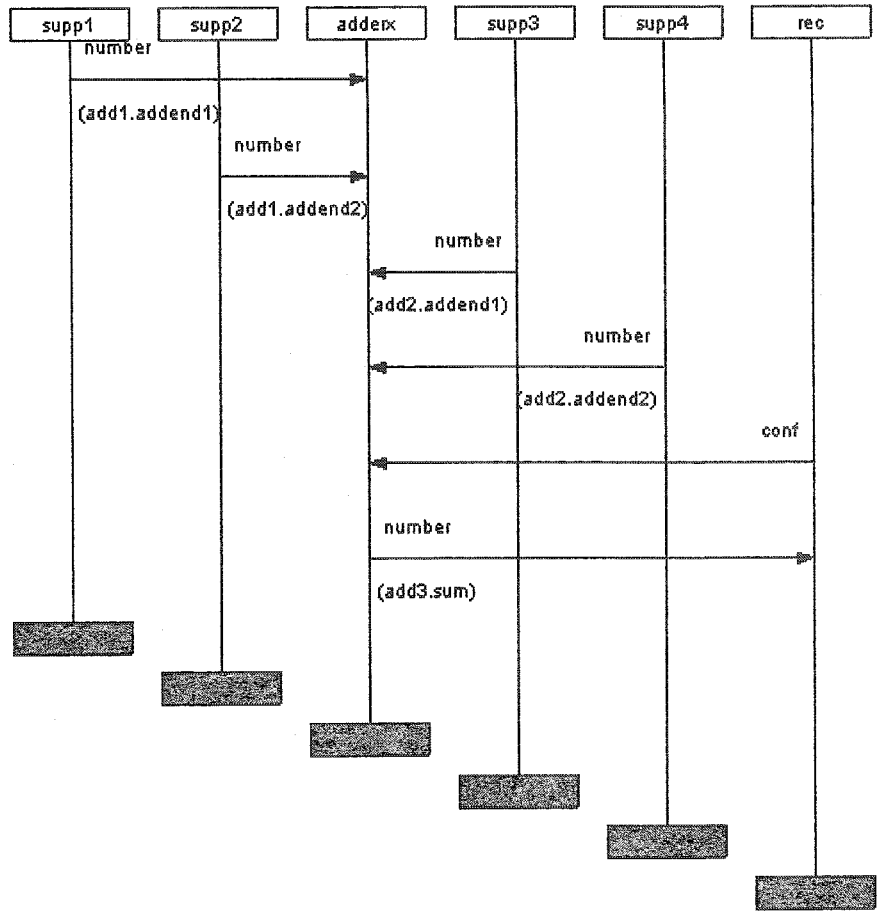
Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp3: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp3;
supp4: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp4;
adderx: out number,5(add3.sum) to rec;
rec: in number,5(add3.sum) from adderx;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
	

The third configuration for control flow oriented testing:

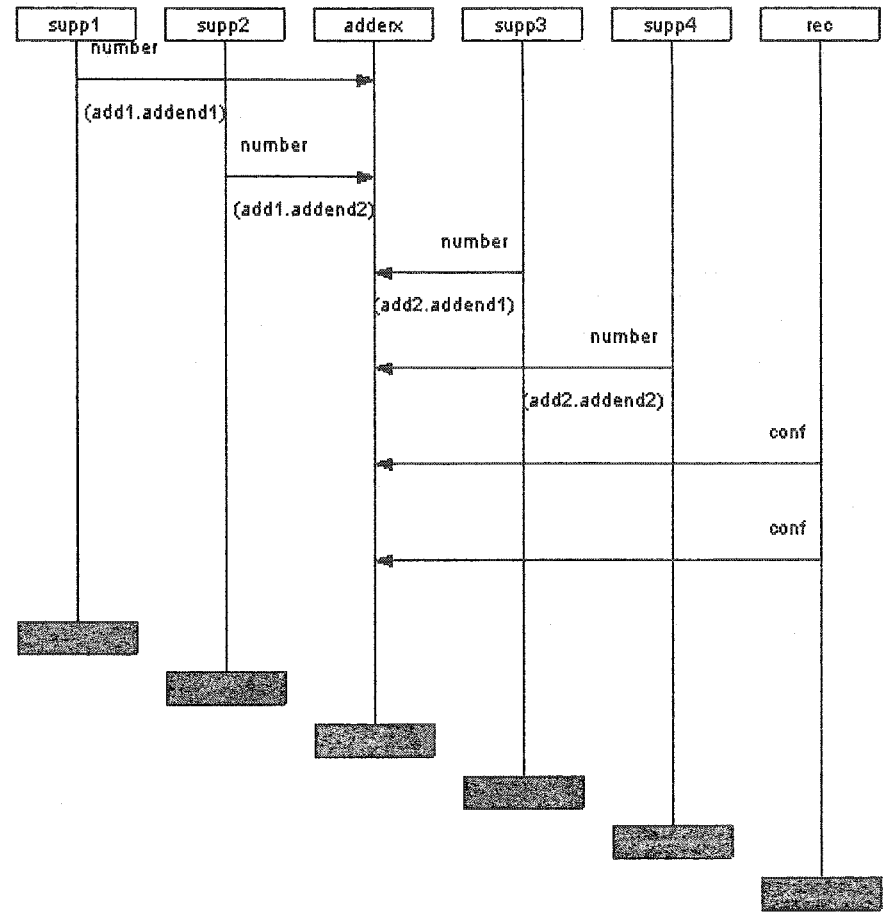
Textual format MSC	Graphical format MSC

```
msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp3: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp3;
supp4: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp4;
rec: out conf,5 to adderx;
adderx: in conf,5 from rec;
adderx: out number,6(add3.sum) to rec;
rec: in number,6(add3.sum) from adderx;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
```



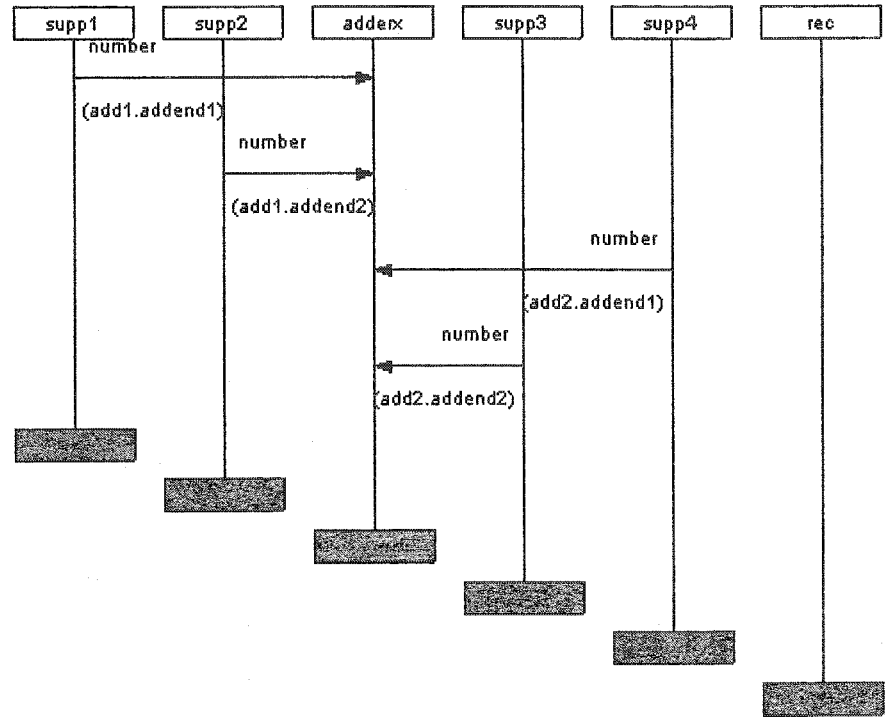
The fourth configuration for control flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp3: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp3;
supp4: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp4;
rec: out conf,5 to adderx;
adderx: in conf,5 from rec;
rec: out conf,6 to adderx;
adderx: in conf,6 from rec;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
	

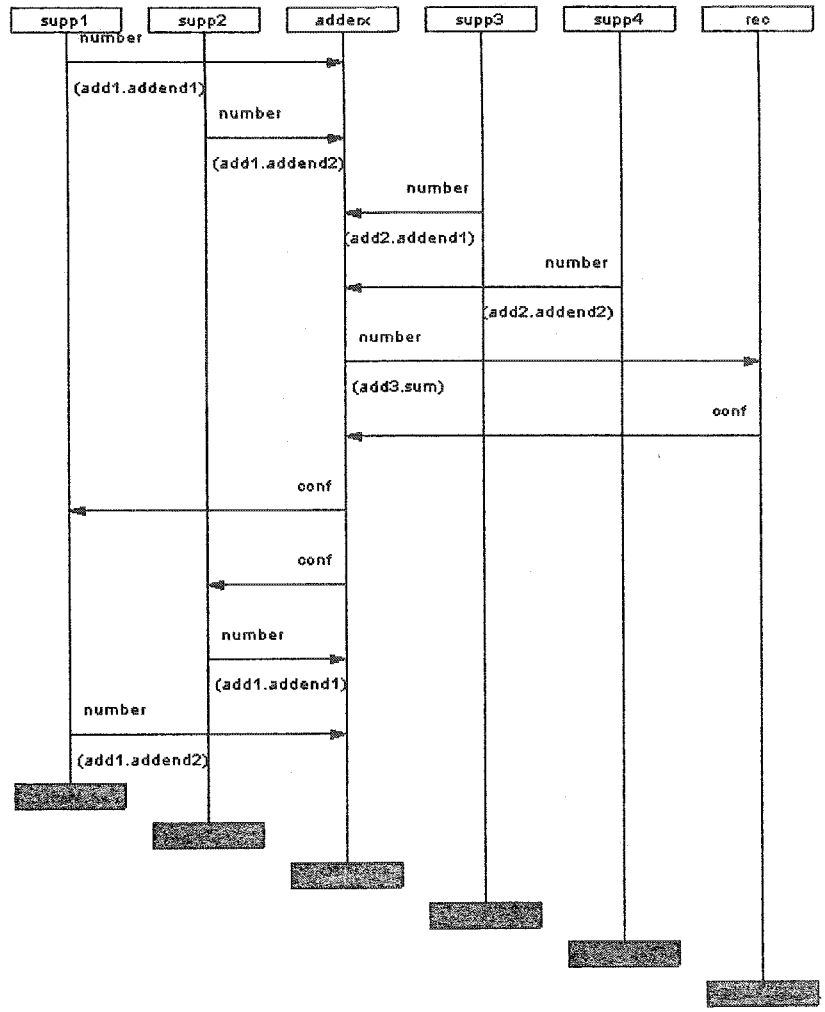
The fifth configuration for control flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp4: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp4;
supp3: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp3;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
	

The sixth configuration for control flow oriented testing:

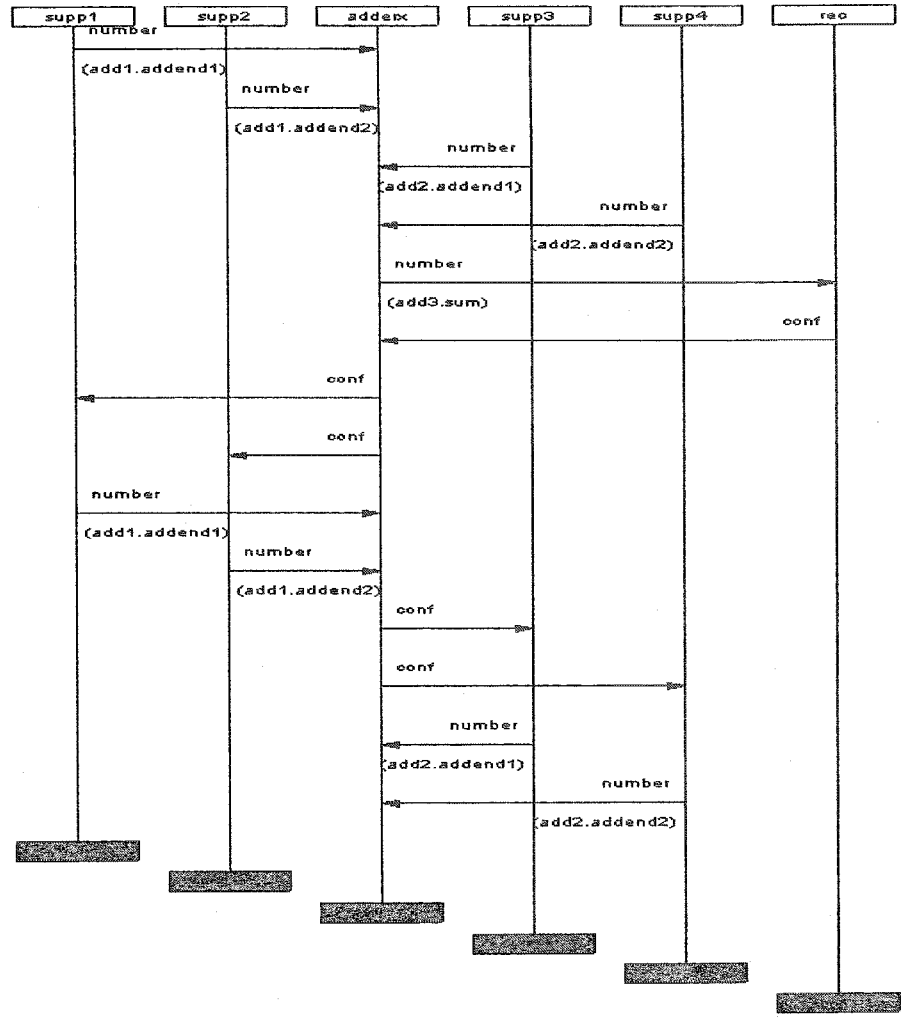
Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp3: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp3;
supp4: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp4;
adderx: out number,5(add3.sum) to rec;
rec: in number,5(add3.sum) from adderx;
rec: out conf,6 to adderx;
adderx: in conf,6 from rec;
adderx: out conf,7 to supp1;
supp1: in conf,7 from adderx;
adderx: out conf,8 to supp2;
supp2: in conf,8 from adderx;
supp2: out number,9(add1.addend1) to adderx;
adderx: in number,9(add1.addend1) from supp2;
supp1: out number,10(add1.addend2) to adderx;
adderx: in number,10(add1.addend2) from supp1;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
	

The seventh configuration for control flow oriented testing:

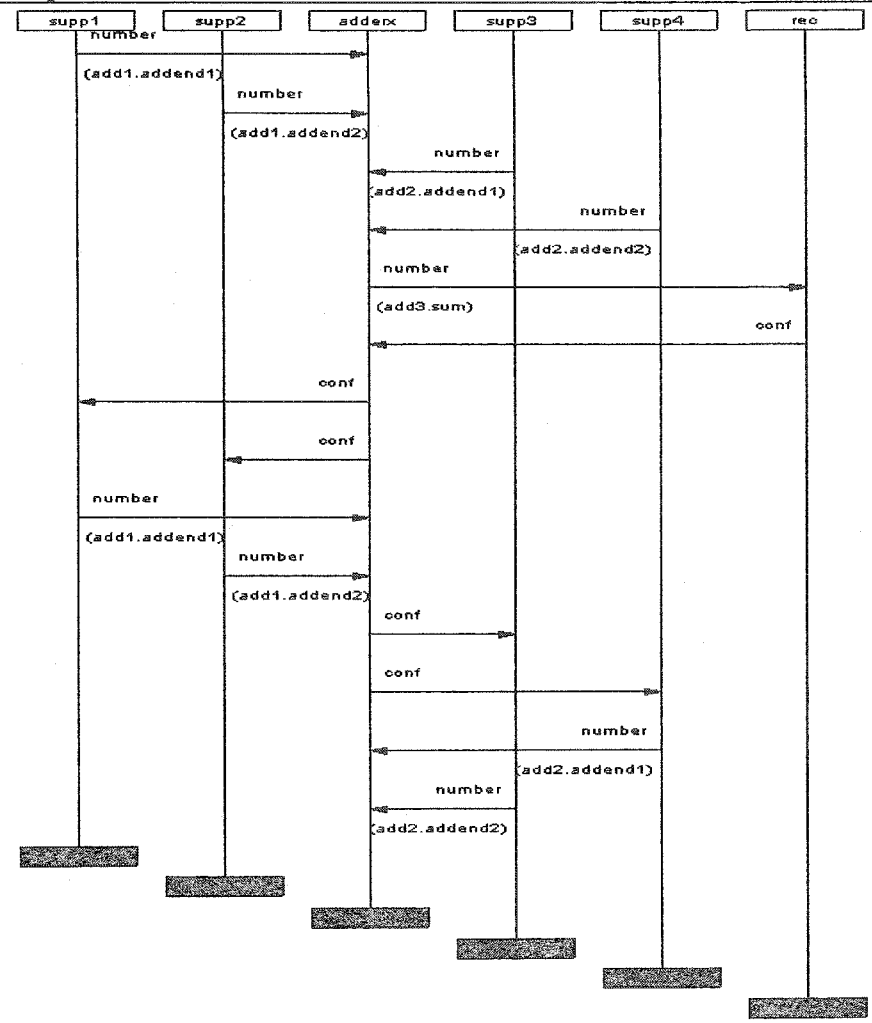
Textual format MSC	Graphical format MSC

```
msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp3: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp3;
supp4: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp4;
adderx: out number,5(add3.sum) to rec;
rec: in number,5(add3.sum) from adderx;
rec: out conf,6 to adderx;
adderx: in conf,6 from rec;
adderx: out conf,7 to supp1;
supp1: in conf,7 from adderx;
adderx: out conf,8 to supp2;
supp2: in conf,8 from adderx;
supp1: out number,9(add1.addend1) to adderx;
adderx: in number,9(add1.addend1) from supp1;
supp2: out number,10(add1.addend2) to adderx;
adderx: in number,10(add1.addend2) from supp2;
adderx: out conf,11 to supp3;
supp3: in conf,11 from adderx;
adderx: out conf,12 to supp4;
supp4: in conf,12 from adderx;
supp3: out number,13(add2.addend1) to adderx;
adderx: in number,13(add2.addend1) from supp3;
supp4: out number,14(add2.addend2) to adderx;
adderx: in number,14(add2.addend2) from supp4;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
```



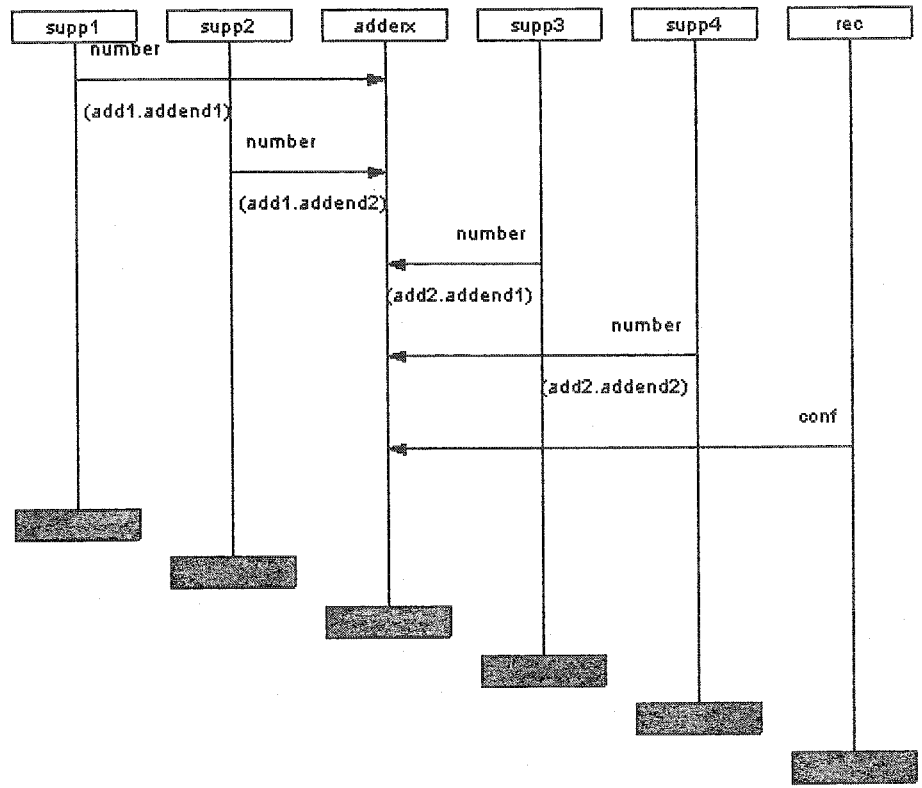
The eighth configuration for control flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp3: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp3;
supp4: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp4;
adderx: out number,5(add3.sum) to rec;
rec: in number,5(add3.sum) from adderx;
rec: out conf,6 to adderx;
adderx: in conf,6 from rec;
adderx: out conf,7 to supp1;
supp1: in conf,7 from adderx;
adderx: out conf,8 to supp2;
supp2: in conf,8 from adderx;
supp1: out number,9(add1.addend1) to adderx;
adderx: in number,9(add1.addend1) from supp1;
supp2: out number,10(add1.addend2) to adderx;
adderx: in number,10(add1.addend2) from supp2;
adderx: out conf,11 to supp3;
supp3: in conf,11 from adderx;
adderx: out conf,12 to supp4;
supp4: in conf,12 from adderx;
supp4: out number,13(add2.addend1) to adderx;
adderx: in number,13(add2.addend1) from supp4;
supp3: out number,14(add2.addend2) to adderx;
adderx: in number,14(add2.addend2) from supp3;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
	

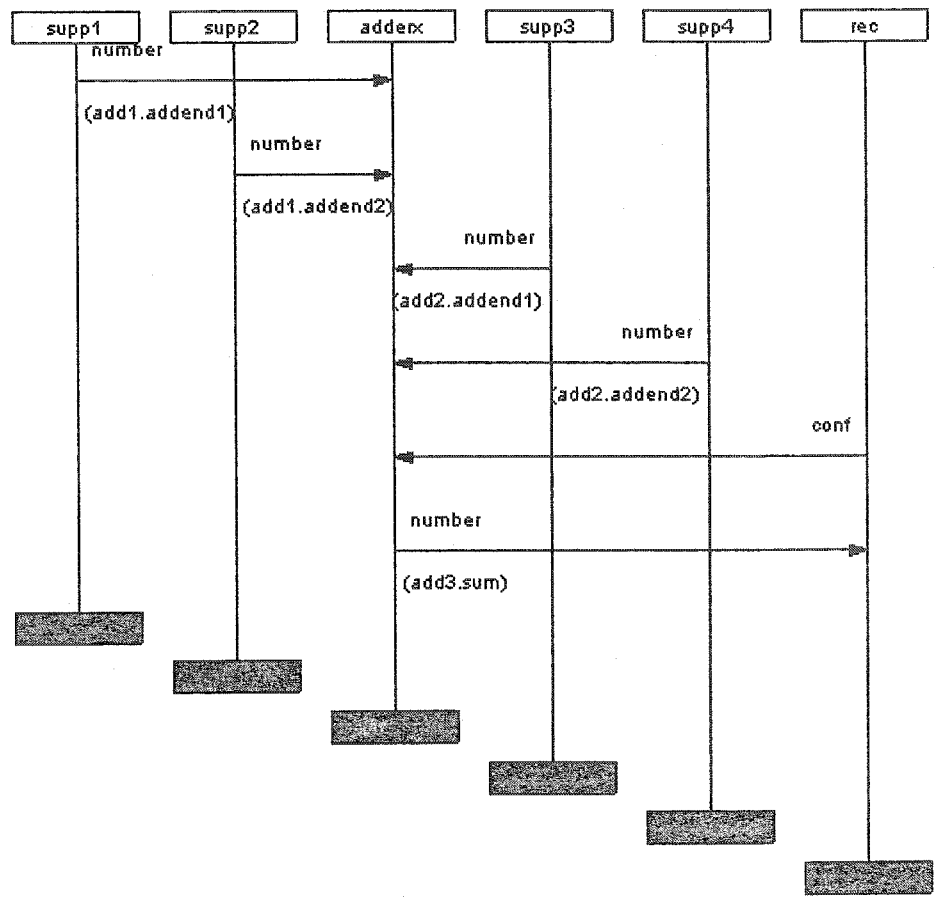
The nineth configuration for control flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp3: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp3;
supp4: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp4;
rec: out conf,5 to adderx;
adderx: in conf,5 from rec;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
	

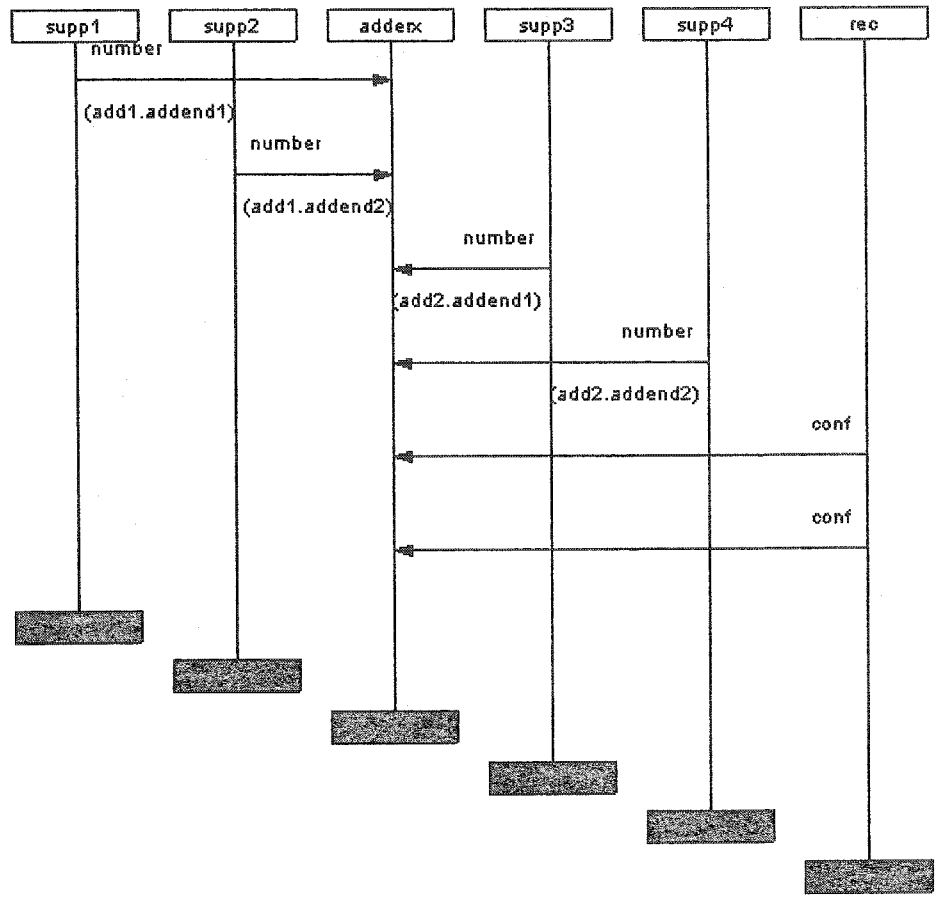
The tenth configuration for control flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp3: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp3;
supp4: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp4;
rec: out conf,5 to adderx;
adderx: in conf,5 from rec;
adderx: out number,6(add3.sum) to rec;
rec: in number,6(add3.sum) from adderx;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
	

The eleventh configuration for control flow oriented testing:

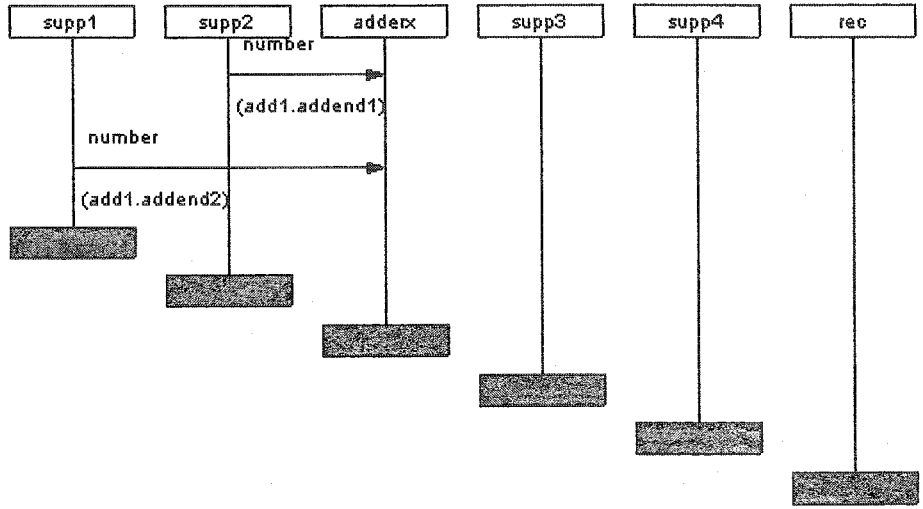
Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp3: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp3;
supp4: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp4;
rec: out conf,5 to adderx;
adderx: in conf,5 from rec;
rec: out conf,6 to adderx;
adderx: in conf,6 from rec;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
	

B.3 Configurations for Data Flow Oriented Testing

The first configuration for data flow oriented testing:

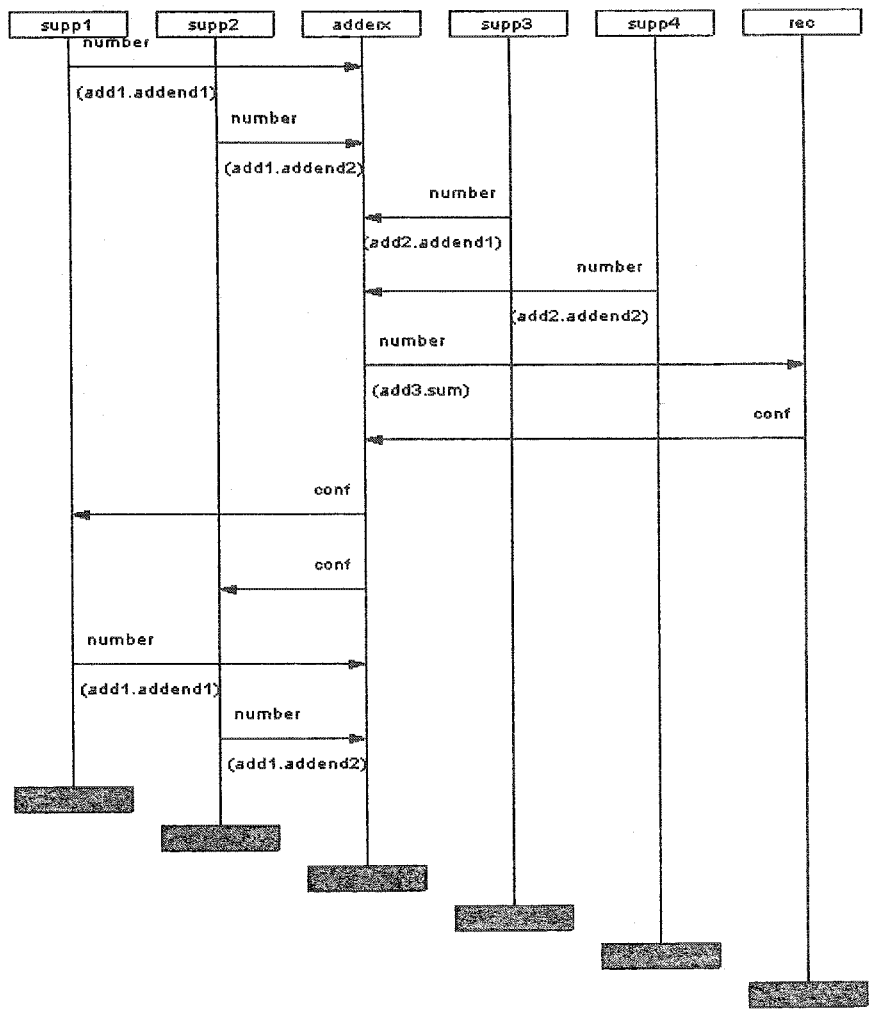
Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp2: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp2;
supp1: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp1;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
	

The second configuration for data flow oriented testing:

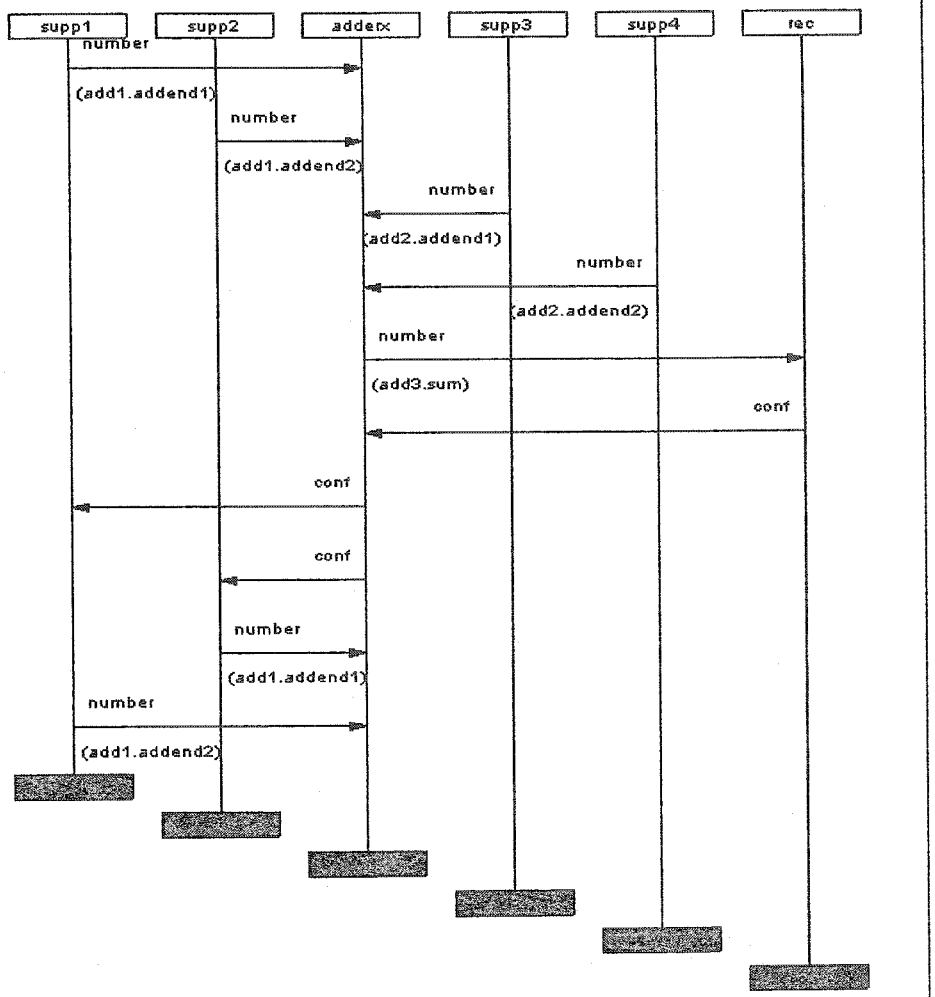
Textual format MSC	Graphical format MSC

```
msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp3: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp3;
supp4: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp4;
adderx: out number,5(add3.sum) to rec;
rec: in number,5(add3.sum) from adderx;
rec: out conf,6 to adderx;
adderx: in conf,6 from rec;
adderx: out conf,7 to supp1;
supp1: in conf,7 from adderx;
adderx: out conf,8 to supp2;
supp2: in conf,8 from adderx;
supp1: out number,9(add1.addend1) to adderx;
adderx: in number,9(add1.addend1) from supp1;
supp2: out number,10(add1.addend2) to adderx;
adderx: in number,10(add1.addend2) from supp2;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
```



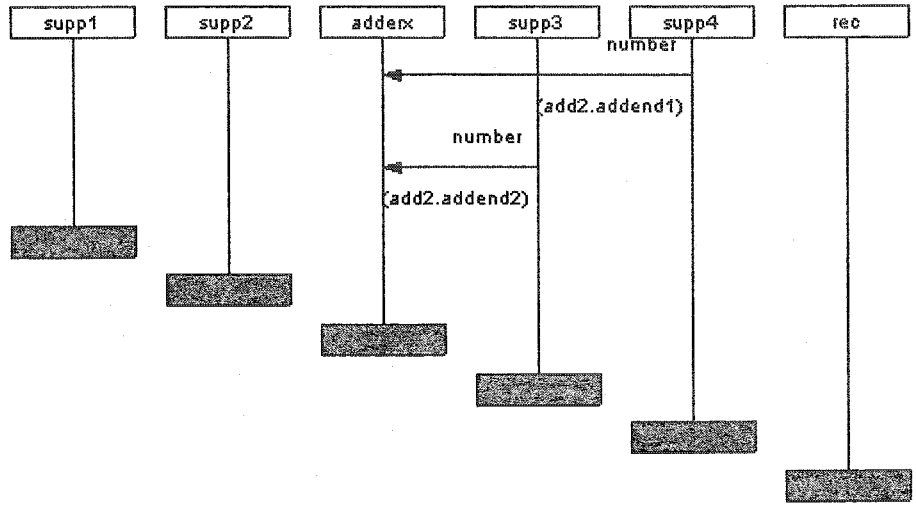
The third configuration for data flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp3: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp3;
supp4: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp4;
adderx: out number,5(add3.sum) to rec;
rec: in number,5(add3.sum) from adderx;
rec: out conf,6 to adderx;
adderx: in conf,6 from rec;
adderx: out conf,7 to supp1;
supp1: in conf,7 from adderx;
adderx: out conf,8 to supp2;
supp2: in conf,8 from adderx;
supp2: out number,9(add1.addend1) to adderx;
adderx: in number,9(add1.addend1) from supp2;
supp1: out number,10(add1.addend2) to adderx;
adderx: in number,10(add1.addend2) from supp1;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
	

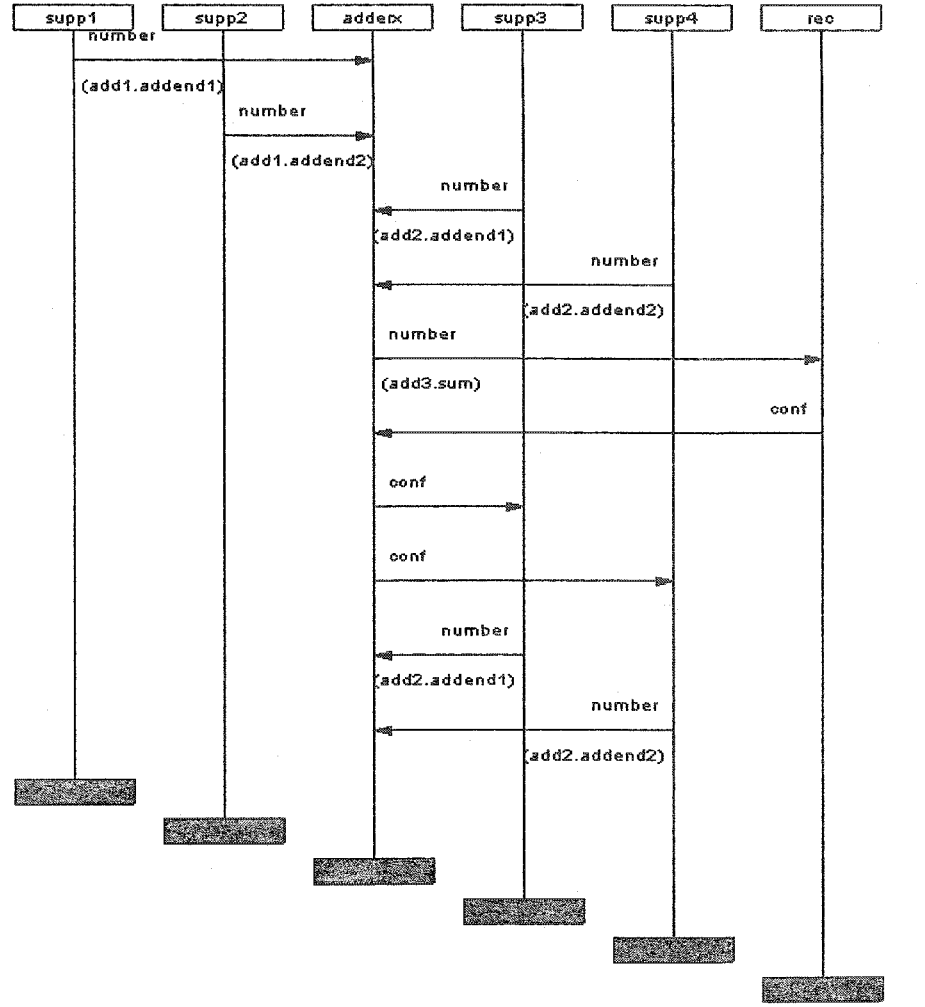
The fourth configuration for data flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp4: out number,1(add2.addend1) to adderx;
adderx: in number,1(add2.addend1) from supp4;
supp3: out number,2(add2.addend2) to adderx;
adderx: in number,2(add2.addend2) from supp3;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
	

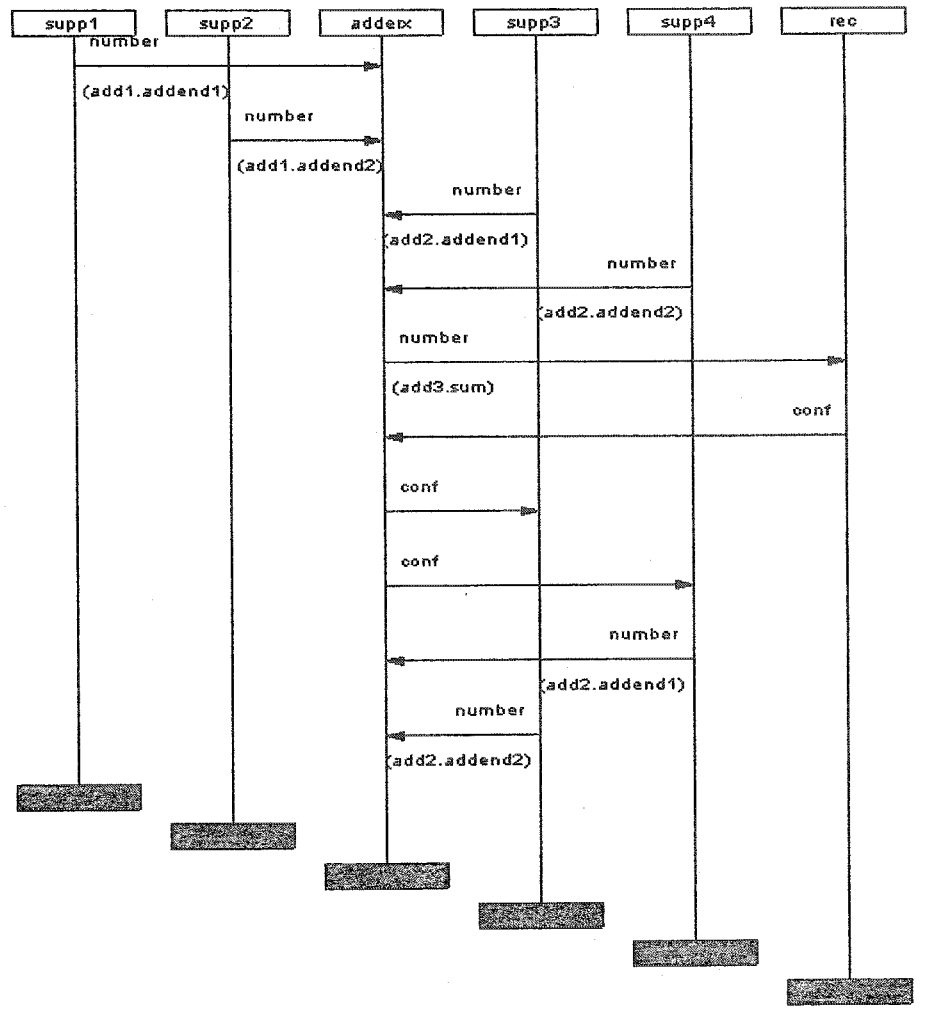
The fifth configuration for data flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp3: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp3;
supp4: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp4;
adderx: out number,5(add3.sum) to rec;
rec: in number,5(add3.sum) from adderx;
rec: out conf,6 to adderx;
adderx: in conf,6 from rec;
adderx: out conf,7 to supp3;
supp3: in conf,7 from adderx;
adderx: out conf,8 to supp4;
supp4: in conf,8 from adderx;
supp3: out number,9(add2.addend1) to adderx;
adderx: in number,9(add2.addend1) from supp3;
supp4: out number,10(add2.addend2) to adderx;
adderx: in number,10(add2.addend2) from supp4;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
	

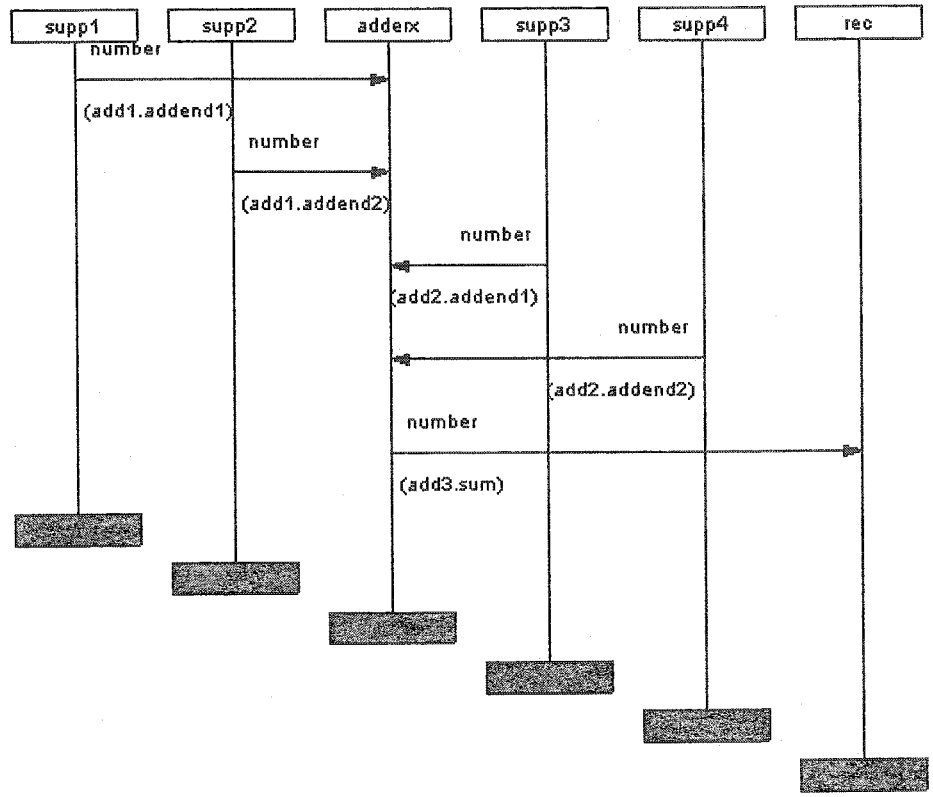
The sixth configuration for data flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp3: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp3;
supp4: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp4;
adderx: out number,5(add3.sum) to rec;
rec: in number,5(add3.sum) from adderx;
rec: out conf,6 to adderx;
adderx: in conf,6 from rec;
adderx: out conf,7 to supp3;
supp3: in conf,7 from adderx;
adderx: out conf,8 to supp4;
supp4: in conf,8 from adderx;
supp4: out number,9(add2.addend1) to adderx;
adderx: in number,9(add2.addend1) from supp4;
supp3: out number,10(add2.addend2) to adderx;
adderx: in number,10(add2.addend2) from supp3;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
	

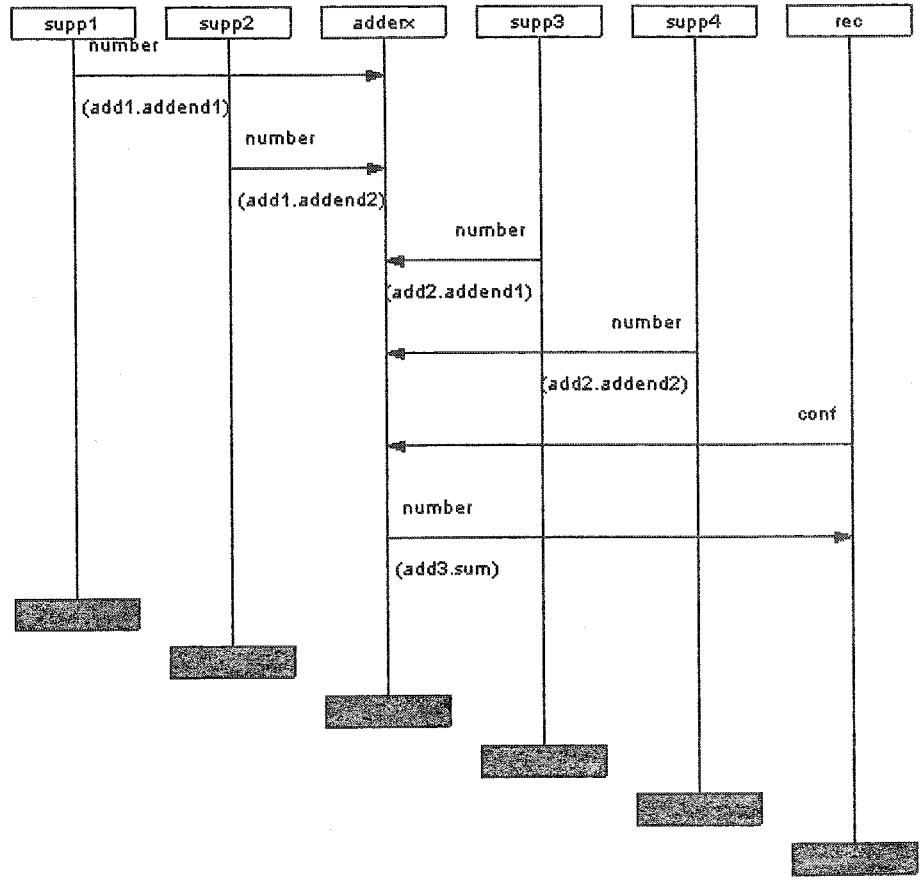
The seventh configuration for data flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp3: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp3;
supp4: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp4;
adderx: out number,5(add3.sum) to rec;
rec: in number,5(add3.sum) from adderx;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
	

The eighth configuration for data flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp3: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp3;
supp4: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp4;
rec: out conf,5 to adderx;
adderx: in conf,5 from rec;
adderx: out number,6(add3.sum) to rec;
rec: in number,6(add3.sum) from adderx;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;
	

The ninth configuration for data flow oriented testing:

Textual format MSC	Graphical format MSC

msc configuration;
supp1: instancehead;
supp2: instancehead;
adderx: instancehead;
supp3: instancehead;
supp4: instancehead;
rec: instancehead;
supp1: out number,1(add1.addend1) to adderx;
adderx: in number,1(add1.addend1) from supp1;
supp2: out number,2(add1.addend2) to adderx;
adderx: in number,2(add1.addend2) from supp2;
supp3: out number,3(add2.addend1) to adderx;
adderx: in number,3(add2.addend1) from supp3;
supp4: out number,4(add2.addend2) to adderx;
adderx: in number,4(add2.addend2) from supp4;
rec: out conf,5 to adderx;
adderx: in conf,5 from rec;
adderx: out number,6(add3.sum) to rec;
rec: in number,6(add3.sum) from adderx;
supp1: endinstance;
supp2: endinstance;
adderx: endinstance;
supp3: endinstance;
supp4: endinstance;
rec: endinstance;
endmsc;