

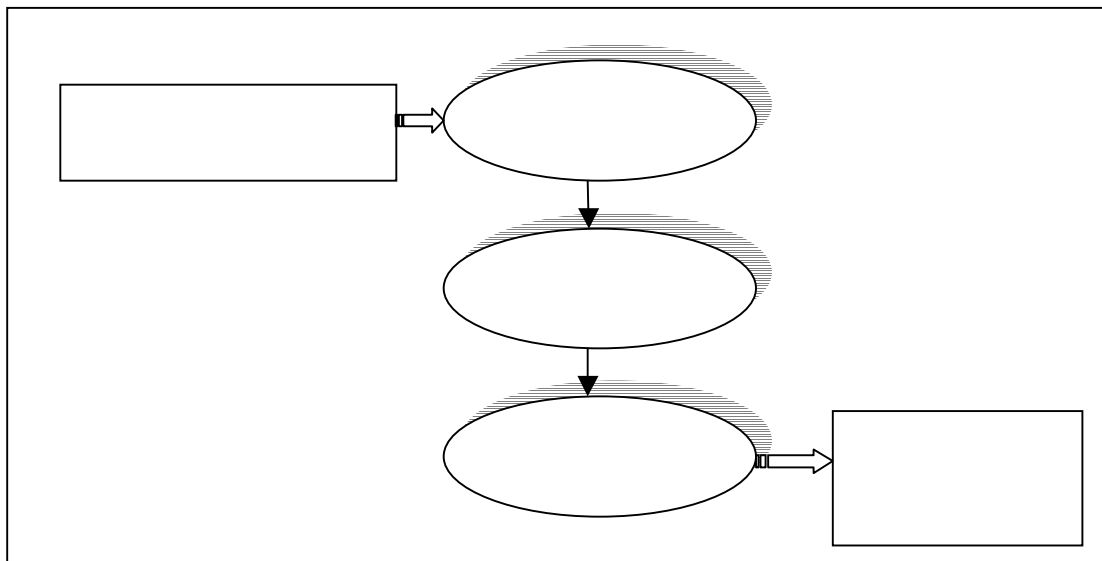
# LES2MSC

This prototype 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 (LES) [1] representation of a system of asynchronously communicating extended finite state machines (EFSMs). The prototype consists of three major parts, which are shown in Figure 1:

Phase 1: Processing an LES

Phase 2: Configuration Construction

Phase 3: Test Construction



**Figure 1 The Software Configuration**

The process performed by the prototype 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 put 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 [2] or data flow oriented testing [3]. Third, the configurations are put in textual MSC [4] 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.

## ***Input File Format***

The input of the prototype is a labeled event structure representing a system of asynchronously communicating EFSMs.

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

Global state of  $m$  can be represented as  $g = (s_{m_1}, \dots, s_{m_n}, c_{q_1}, \dots, c_{q_r})$ , where

- $s_{m_1} \dots s_{m_n}$  are the current states of the asynchronously communicating EFSMs  $m_1 \dots m_n$  and
- $c_{q_1} \dots c_{q_r}$  are the contents of the queues between the EFSMs.

A Labeled Event Structure (LES)  $\sigma$ , 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 research 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  $\sigma$  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  $\sigma$ .

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  $\sigma$  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  $\omega = v_1 \dots v_p$ , the prefix contains a configuration  $C' = C \cup \{e, e_1, \dots, e_p\}$  with  $e, e_1, \dots, e_p \notin C$  and  $l(e) = \mu$ ,  $l(e_1) = v_1, \dots, l(e_p) = v_p$ .

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

The input file representing an LES 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  $E$  is

“label(et2)=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  $S$  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 is:

“et0<et1<et2;”.

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

It has the following format:

eventid '#' eventid ';'.

An example of conflicting events is:

“et3#et6;”.

-  $C$  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 is:

“{et9, em9, er3} -> {et0, em0, er0};”. In this example, the corresponding global state of cutoff point “et9, em9, er3” is “et0, em0, er0”.

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 “swpx.les” is given in Appendix A.

### ***How to run the executable file***

1. Copy the executable file “[www.site.uottawa.ca/~ural/LES2MSC/Objectcode/les2msc](http://www.site.uottawa.ca/~ural/LES2MSC/Objectcode/les2msc)” to your own directory.
2. Change its attribute to “executable”. (This step can be done in Solaris environment by typing command: “`chmod 777 les2msc`”).
3. Copy “.les” file to the same directory. (Examples “swpx.les” and “adderbak.les” can be found in “[www.site.uottawa.ca/~ural/LES2MSC/Docs&Examples/](http://www.site.uottawa.ca/~ural/LES2MSC/Docs&Examples/)”).
4. Open X-Window, choose Solaris environment, and go to the directory which contains the executable “les2msc” file and the “.les” file.

5. Type in the command line “les2msc filename”. (filename is the “.les” file name)  
Then the output “.mpr” files are obtained in the same directory.

## ***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. The syntax and semantics of MSCs are standardized by ITU-T, as recommendation Z.120 [4].

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.

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 2.

MSC configuration

MSC generated by les2msc

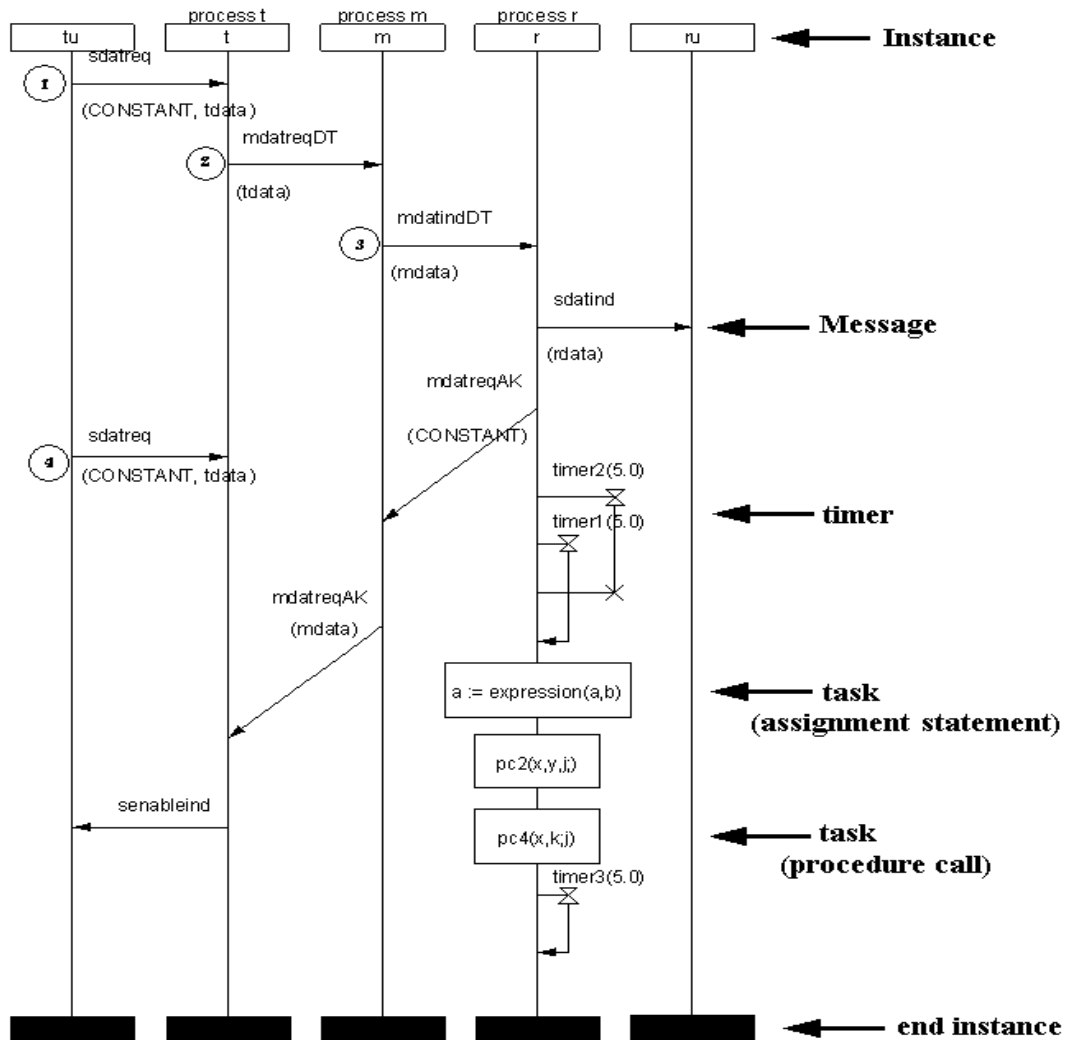


Figure 2 An MSC Example

As shown in Figure 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 MSCs generated by this prototype for example “swpx.les” are given in Appendix B.

## **References**

- [1] O. Henniger, “Test generation from specifications in Estelle and SDL”, Brandenburg University of Technology Cottbus, Germany, PhD thesis, 2002, in German
- [2] G.J. Myers, The art of software testing. New York: John Wiley & Sons, 1979
- [3] 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
- [4] ITU Recommendation Z.120, Message Sequence Charts (MSC).

## **Appendix A**      ***Input file of Example “swpx.les”***

The input LES file for example “swpx.les” is as follows:

---

```
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};

---

## Appendix B Output MSCs of Example “swpx.les”

### Configurations for Control Flow Oriented Testing

The first configuration for control flow oriented testing:

Textual format MSC	Graphical format MSC
<pre> 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; </pre>	<p>The graphical format MSC diagram shows three lifelines: tu, swpx, and ru. The sequence of messages is as follows:</p> <ul style="list-style-type: none"> <li>tu sends sdatreq to swpx with parameters (CONSTANT, t.data).</li> <li>swpx sends sdatind to ru with parameter (r.data).</li> <li>tu sends sdatreq to swpx with parameters (CONSTANT, t.data).</li> <li>swpx sends sdisableind to tu.</li> <li>swpx sends senableind to tu.</li> <li>swpx sends sdatind to ru with parameter (r.data).</li> </ul>

The second configuration for control flow oriented testing:

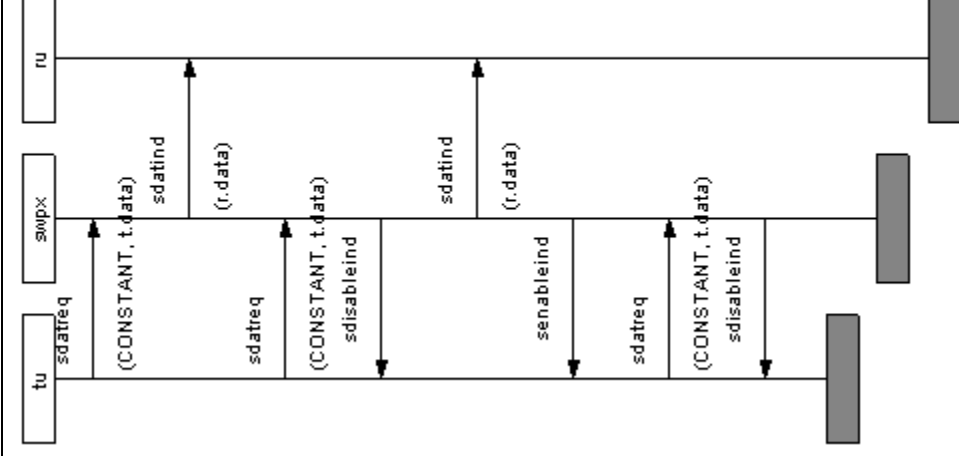
Textual format MSC

```

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;

```

Graphical format MSC



## Configurations for Data Flow Oriented Testing

The first configuration for data flow oriented testing:

Textual format MSC	Graphical format MSC
<pre> 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; </pre>	<p>The diagram illustrates the sequence of messages between three lifelines: tu, swpx, and ru.      1. tu sends sdatreq to swpx with parameter (CONSTANT, t.data).     2. swpx sends sdatind to ru with parameter (r.data).     3. tu sends sdatreq to swpx with parameter (CONSTANT, t.data).     4. swpx sends sdisableind to tu.     5. swpx sends senableind to tu.     6. tu sends sdatreq to swpx with parameter (CONSTANT, t.data).     Lifelines for tu, swpx, and ru are shown as boxes at the top, with vertical lines representing their duration. Shaded rectangular blocks are placed on the lifelines to indicate activation periods during the message exchanges.</p>

The second configuration for data flow oriented testing:

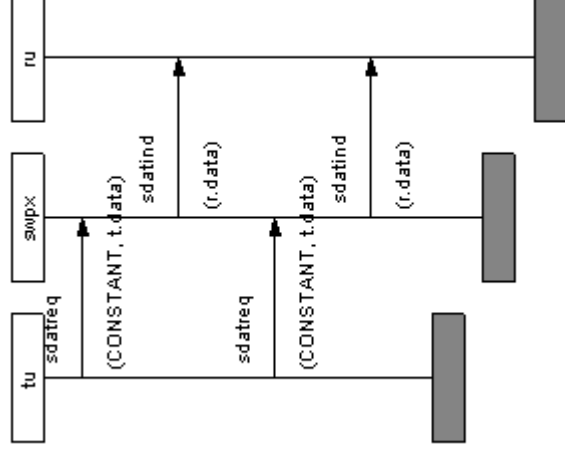
Textual format MSC

```

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;

```

Graphical format MSC



The third configuration for data flow oriented testing:

Textual 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;

```

Graphical format MSC

