KLOCwork
MSC to SDL Synthesizer
Reference Manual

Release 2.0

KLOCwork

Printed on 9 April, 2002

KLOCwork Corporation
Toll-free telephone: 1-866-556-2967
E-mail:
sales@klocwork.com
support@klocwork.com
Website: http://www.klocwork.com
Corporate Headquarters:
1 Antares Drive, Suite 510
Ottawa, Ontario
Canada
K2E 8C4
(613) 224-2277
US Headquarters:
1700 Montgomery Street
Suite 111
San Francisco, CA
94111
(415) 954-7154

Contents

CHAPTER 1

Introduction

In This Chapter

What is the KLOCwork MSC to SDL Synthesizer?

The KLOCwork MSC to SDL Synthesizer, a member of the KLOCwork Bridge family of products, takes a set of Message Sequence Charts (MSC) that comprise a scenario model and *automatically transforms the scenario model into an executable state machine model* in Specification and Description Language (SDL) that conforms to standards and is ready to use with SDL tools such as Telelogic Tau.

This document contains information that we hope will assist you in day-to-day use of KLOCwork MSC to SDL Synthesizer. This document contains the following parts:

- ***MSC and SDL notations*** (on page 13) provides an overview of the input and output languages of the KLOCwork MSC to SDL Synthesizer.
- ***Installing MSC to SDL Synthesizer*** on page 21 explains how to install the KLOCwork MSC to SDL Synthesizer.
- ***Running the KLOCwork MSC to SDL Synthesizer*** (on page 23) describes how to use the Synthesizer integrated with Telelogic Tau or standalone.
- ***Synthesis sections*** (see "Synthesis: Basic MSC scenarios" on page 39) describe the graphical representation of the MSC language, describe how the KLOCwork MSC to SDL Synthesizer interprets input MSCs, and describe what SDL constructs are produced.
- ***MSC PR syntax*** (on page 99) defines the textual syntax of the subset of MSC language supported by the KLOCwork MSC to SDL Synthesizer.
- ***Error messages (MSC to SDL)*** (see "Error messages" on page 107) lists error messages associated with KLOCwork MSC to SDL Synthesizer.

To learn more about the concepts and operations related to the Synthesizer, refer to *KLOCwork MSC to SDL Synthesizer Tutorial*. For more examples of MSC specifications, see the *KLOCwork MSC to SDL Synthesizer Cookbook of MSC Specifications*.

Bridging scenarios and state machine models

Today there exists a discontinuity between scenarios and state machines. Bridging these two types of formal models by automatically synthesizing executable state machines from scenarios allows rapid prototyping of products using scenarios rather than state machines.

There are important advantages of moving beyond state machine modeling towards scenario modeling. Scenarios are used at very early phases of the development process when requirements for the future product are being captured and analyzed. Scenarios are simple enough to be used in discussions with various stakeholders of the development. Scenarios are flexible enough to be easily modified and updated which is vital for lucid and creative processes at early development phases.

On the other hand, state machines are currently the established starting point for formal modeling. State machines are more expressive and powerful than scenarios, but they require more effort to produce, understand, and maintain.

Automatic synthesis of state machines from scenarios moves the benefits of formal, tool-supported modeling and validation to earlier phases of the development process, thereby stopping the snowball of incorrect decisions driven by incomplete or incorrect requirements. This helps get your product to market faster while ensuring that you meet customer expectations.

KLOCwork Bridge family of products

The KLOCwork Bridge family of products provides a springboard for developing software components using scenarios. These products focus on the early phases of the software development process, particularly requirements capture and validation, architecture definition and validation, and system testing. The KLOCwork Bridge family of products allows use of a scenario language during the modeling process and automatically transforms scenarios into executable state machines.

The architecture of the KLOCwork Bridge product family consists of the following components:

Scenario Analyzer performs syntax and semantic analysis of the input scenario model.

Synthesizer creates an intermediate representation of the executable state machine model in the form of Event Automata.

Code Generator takes the output of the Synthesizer and produces the particular syntax of the language to which you want to bridge.

MSC to SDL Synthesizer

The KLOCwork MSC to SDL Synthesizer takes a scenario model described as a set of the ITU standard Message Sequence Charts (MSC) and automatically transforms the scenario model into an executable state machine model in the ITU standard Specification and Description Language (SDL). The generated model is ready to use with SDL tools such as Telelogic Tau.

Here is a brief overview of how MSC constructs are *mapped* to SDL:

- Collection of multiple MSCs is mapped onto a single SDL system
- MSC instances are mapped onto SDL processes
- MSC message names are mapped onto SDL signals
- MSC message output events are mapped to signal outputs in an SDL process
- MSC message input events are mapped to signal consumption stimuli in an SDL state
- MSC timer set, reset, and timeout events are mapped to setting a timer, explicit timer reset, and consumption of timer signal respectively
- instance stop is mapped to process termination

The KLOCwork MSC to SDL Synthesizer creates SDL models using *two types of code generators*: the plain SDL generator and the type-based SDL generator. The plain SDL generator creates the simplest form of SDL model. The type-based SDL generator creates models using types, inheritance, and virtual transitions packaged in a way that allows easy reuse. The type-based SDL is meant for manual modifications and updates by using the language capabilities of SDL without affecting the generated portion.

MSC to SDL Synthesizer in the development process

The KLOCwork MSC to SDL Synthesizer is a versatile tool. The following paragraphs illustrate how it can add value in the development process.

Use MSCs to jumpstart your SDL project

The KLOCwork MSC to SDL Synthesizer can be used to jumpstart your SDL project. Start with few easy-to-understand scenarios, then simply click a button and the Synthesizer automatically generates a complete SDL specification ready to run in an SDL tool.

1. Start Telelogic Tau.
2. Use your Telelogic MSC Editor (MSCE) to create MSC diagrams.
3. Use the Telelogic HMSC Editor (HMSCE) to create High-level MSC diagrams.
4. Run the KLOCwork MSC to SDL Synthesizer to produce an SDL model that corresponds to your HMSC model.
5. Use your SDL tool to review and simulate the model.
6. Refine the MSC model.
7. Continue SDL project by refining the generated SDL model.

Use MSCs to rapidly prototype requirements

Jumping into coding too early and not understanding the requirements can result in developing a product that does not meet customer expectations. By using scenarios, you can describe functional requirements in an easily understood format that can be discussed with non-technical stakeholders.

1. Use Telelogic MSC and HMSC Editors to capture functional requirements as scenarios that focus on the interaction between your system and its environment.
2. Simulate MSCs by using the KLOCwork MSC to SDL Synthesizer and Telelogic Simulator UI.
3. When you notice problem areas, go back to your MSC tool, add more scenarios to solve the problems. Rerun the Synthesizer to get the new SDL specification.
4. Repeat this process until you are satisfied with your prototype (scenarios and SDL model).

Use MSCs to perform automatic early fault detection

Even when the goals of your project do not include building an SDL model, the KLOCwork MSC to SDL Synthesizer can be used for early fault detection in you MSC models. It can dramatically extend the value of your MSC tools by adding the capability of automatic validation techniques, for example, those provided by the Telelogic Tau Validator.

The automatically synthesized SDL model can be imported into an SDL tool and analyzed using state-of-the-art state space exploration. The validation process automatically discovers certain faulty behaviors such as deadlocks or failed communication. These faulty behaviors are represented as scenarios using the MSC tool. Anomalies in the behavior of the synthesized model usually reflect problems or inconsistencies in the original MSC model.

1. Use your Telelogic MSC Editor (MSCE) to create MSC diagrams.
2. Use the Telelogic HMSC Editor (HMSCE) to create High-level MSC diagrams.
3. Run the KLOCwork MSC to SDL Synthesizer to produce an SDL model that corresponds to your HMSC model.
4. Use the Telelogic Tau Validator to automatically detect faults in the scenario model.
5. Correct problems by creating additional scenarios.
6. Rerun the KLOCwork MSC to SDL Synthesizer.

Use MSCs to automatically generate test cases

The KLOCwork MSC to SDL Synthesizer can automatically produce test cases as early as the requirements definition phase.

During scenario modeling, the black-box behavior of the system under development is described by specifying desired interactions between the system and its environment. Thus, the scenario model contains both the description of the system and the description of the system's environment.

The KLOCwork MSC to SDL Synthesizer can selectively produce a partial SDL model only for the environment part of the requirements definition. This environment model can be used as a test suite for the subsequent integration testing phase.

Use MSCs for architecture definition and validation

Automatic synthesis can help address the discontinuity between modeling and implementation. By specifying system scenarios that describe collaboration between architectural components and then synthesizing the SDL architectural model, you can achieve the following benefits:

- early architecture validation by simulating system scenarios
- concurrent production of system scenarios for different components for integration by the KLOCwork MSC to SDL Synthesizer

Use MSCs to design components

Once the components of the system have been identified and validated, you can shift the viewpoint of scenario models and use scenarios to define behavior of individual components. Using the powerful data extensions supported by the KLOCwork MSC to SDL Synthesizer, you can use scenarios as a design notation. This provides the following benefits:

- more intuitive designs
- improved collaboration between team members
- ongoing validation of designs by simulation of the synthesized models
- automatic code generation into the implementation language using SDL Code generator

Use KLOCwork MSC to SDL Synthesizer for SDL training

SDL is a complex language. Most people find it easier to understand scenarios than SDL. You can explain a scenario to a new employee then import the scenario into the KLOCwork MSC to SDL Synthesizer to automatically create the corresponding SDL model. Your new employee can learn SDL faster by observing the automatically synthesized SDL for an already familiar scenario.

CHAPTER 2

MSC and SDL notations

In This Chapter

Terms and concepts

The input to the KLOCwork MSC to SDL Synthesizer is a *composition of multiple scenarios with data extensions*. Each scenario describes behavior of one or more *actors*. The composition of scenarios is described by the *composition graph* or roadmap. Scenarios and their compositions are described in the ITU standard Message Sequence Charts (MSC) language.

MSC is a trace language for the specification and description of the communication behavior of system components and their *environment* by means of *message interchange*. Communication behavior in MSCs is presented in a very intuitive and transparent manner, particularly in the graphical environment. Therefore, the MSC language is easy to learn, use, and interpret.

Data extensions to MSC scenarios describe the flows of information through scenarios.

The KLOCwork MSC to SDL Synthesizer automatically produces an executable state machine model in ITU standard Specification and Description Language (SDL) that implements the behavior described by the input MSC model.

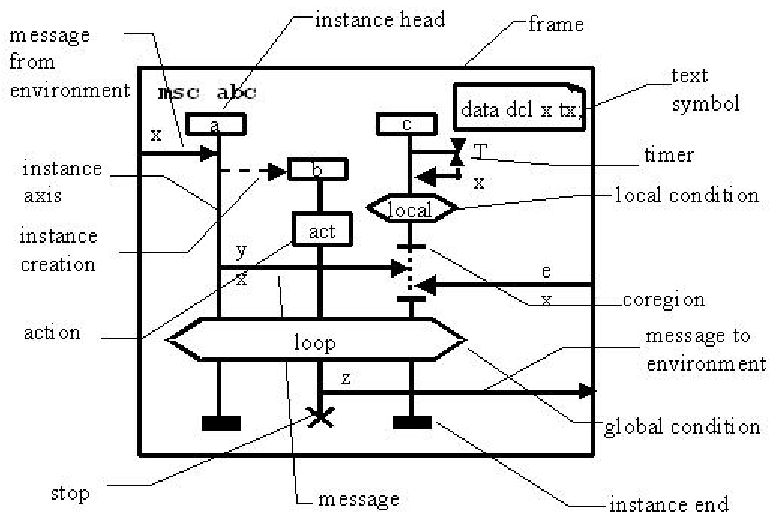
Message Sequence Charts (MSC)

This section provides a brief overview of the Message Sequence Charts specification language. The MSC language supported by the KLOCwork MSC to SDL Synthesizer is based on the MSC-2000 standard. However, certain constructs of the MSC-2000 standard are not supported by the Synthesizer. Data manipulations within MSC scenarios are handled with certain minor deviations from the MSC-2000 standard.

The MSC language has a *graphical notation*. There are two kinds of MSC diagrams: basic message sequence charts (bMSC), and high-level message sequence charts (HMSC). A *textual representation* of MSC specifications is also available. The Telelogic Tau Integration Module of the KLOCwork MSC to SDL Synthesizer allows you to directly use the graphical MSC diagrams from the Telelogic Tau MSC Editor. The input to the core KLOCwork MSC to SDL Synthesizer is several MSCs in textual notation. Telelogic Tau can perform the conversion between the graphical and the textual notations of the MSCs.

The following figure shows a basic MSC. A basic MSC diagram describes the behavior of several *instances*. Each instance is graphically represented as a vertical line (called an *instance axis*). Each instance has an *instance head* that contains the *name* of the instance. An instance axis corresponds to the timeline of the instance. Instances exchange *messages*, which are shown as arrows between two instances or between an instance and the frame of the diagram. An instance can be *created* by another instance. A dashed arrow pointing at the instance head of the child instance shows this parent-child relationship.

Figure 1: Elements of a basic Message Sequence Chart



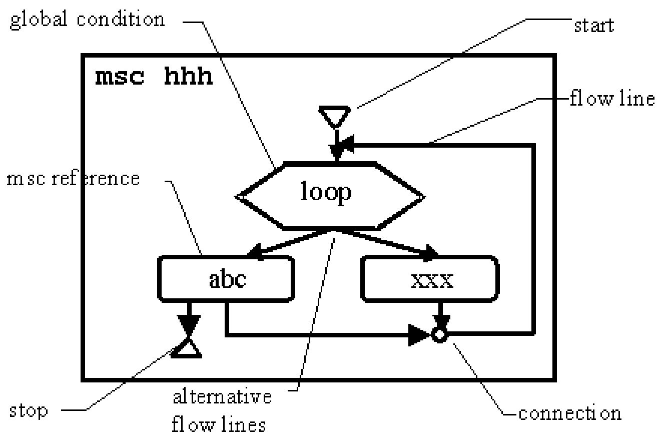
Message Sequence Charts can use *timers*. This figure shows how timer T is *started* by instance c, and then expires, resulting in a *timeout*. An instance can also *stop* a timer (not shown).

A basic MSC describes *events* for each instance. Events on each instance axis are *ordered.* If the first event occurs higher on the instance axis than the second one, the first event occurs *before* the second one. Related pairs of message output and message input introduce another order: message output occurs *before* the corresponding message input. Semantics of an MSC specification is a (transitive) *partial ordering* of the events for all instances in all basic MSC diagrams.

The total ordering of events along each instance in general may not be appropriate. By means of a *coregion,* an exception to this can be made. A coregion is introduced for the specification of unordered events on an instance. Such a coregion in particular covers the practically important case of two or more incoming messages in which the ordering of consumption may be interchanged. For example, the order of consumption for messages y and e on an instance c is not specified (see Elements of a basic Message Sequence Chart diagram).

Composition of basic MSCs is represented graphically using high-level message sequence chart (HMSC) diagrams. As shown in the following figure, an HMSC diagram contains *references* to basic MSC diagrams and *flow lines*.

Figure 2: Elements of a High-level Message Sequence Chart



Data-related elements of MSC specifications include:

- parameters of messages
- parameters of create events
- actions
- local decisions based on data

Composition of basic MSCs can be also described by global *conditions*.

Specification and Description Language (SDL)

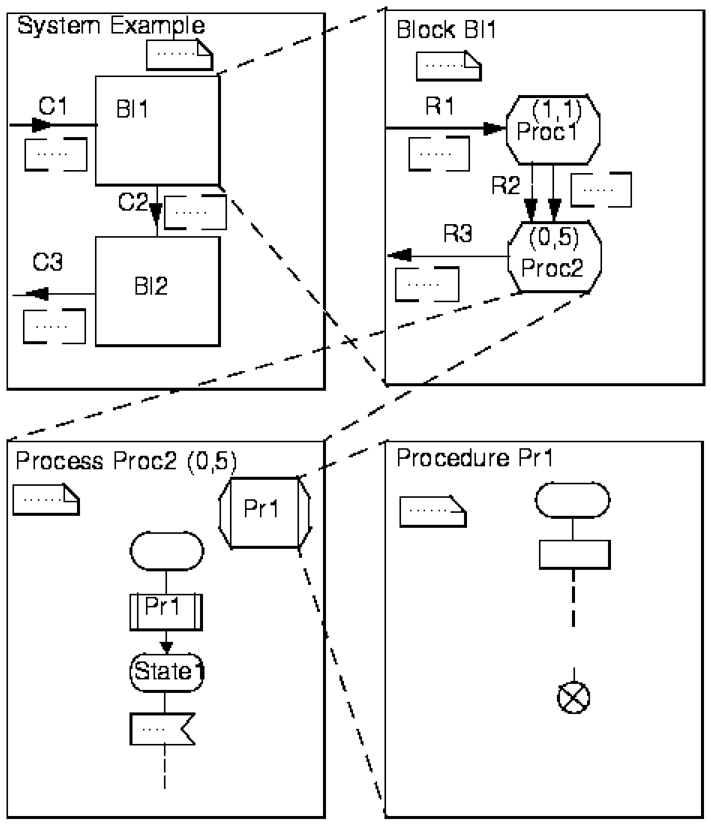
This section provides a brief overview of the Specification and Description Language. The SDL language supported by the KLOCwork MSC to SDL Synthesizer is based on the SDL-96 standard.

The SDL language has both graphical and textual notation. The Telelogic Tau Integration Module of the KLOCwork MSC to SDL Synthesizer represents the generated SDL directly in graphical notation in the Telelogic Tau SDL Editor. The output of the core Synthesizer is SDL in textual notation. Telelogic Tau creates both graphical and textual notations.

Elements of SDL structural diagrams

Let's consider a set of SDL diagrams that follow (Elements of SDL communication).

Figure 3: Elements of SDL structural diagrams



The top left diagram represents an SDL *system* with the name `Example`. The system contains two SDL *blocks* (`Bl1` and `Bl2`). An SDL channel with the name C2 connects the blocks. There are also two other channels: channel C1 connects the *environment* with block `Bl1`, and channel `C3` connects block `Bl2` with the environment.

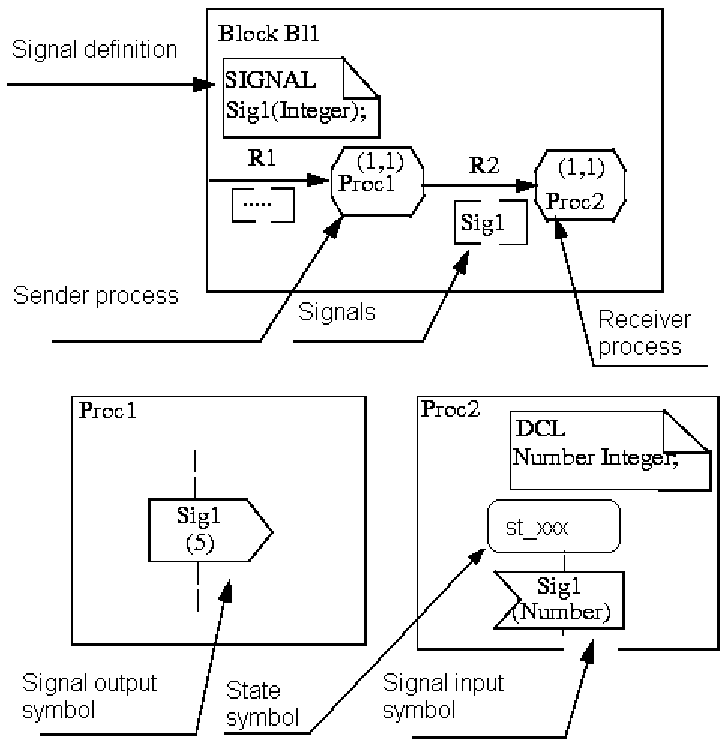
The top right diagram describes the internal structure of the SDL block Bl1. Block Bl1 contains two SDL processes (with names Proc1 and Proc2). Processes are connected by SDL *signalroutes*. Signalroute R1 connects the *environment* of the block Bl1 to process Proc1. Signalroute R2 connects process Proc1 to process Proc2. A dashed line from process Proc1 to process Proc2 indicates that the process Proc1 can *create* instances of the process Proc2. Signalroute R3 connects process Proc2 to the environment of the block Bl1.

The bottom left diagram shows the behavior of the process Proc2. This diagram describes a *state machine* for the process Proc2. The bottom right diagram shows an SDL *procedure*, defined in process Proc2.

Elements of SDL communication

SDL communication is illustrated in the following figure, Elements of SDL communication.

Figure 4: Elements of SDL communication



SDL processes exchange *signals*. Each signal needs to be defined as a *signal definition*. SDL signals can have *parameters*. Each channel and signal route defines the *list of signal names* that can be transferred along this communication path.

In the state machine description, there are symbols for signal *output* and signal *input*. The signal input symbol is always attached to a certain *state* symbol. When the input of a signal with parameters is defined, the parameters of the signal are assigned to *variables*. SDL variables need to be defined separately. A more detailed outline of SDL state machines is provided in ***Elements of SDL state machines*** (on page 18).

Elements of SDL state machines

The following figure illustrates an SDL state machine. This state machine contains a *start* symbol with an empty *transition* into the state WaitOpenDoor. The state WaitOpenDoor has a single *transition* that is initiated by the *input* of signal OpenDoor with a single *parameter (*assigned to *variable* DoorAddr). The transition consists of three *actions*: *output* of the signal Open, *output* of the signal DoorOpened, and *timer start* of the *timer* DoorTimer. Timer DoorTimer is *defined* in the same diagram. The transition enters the *state* Wait_DoorTimer.

The state Wait_DoorTimer describes how the timeout event from the timer DoorTimer is handled. The transition is initiated by the input of the signal from timer DoorTimer and consists of a single action – output of the signal Close with a single parameter. The current value of the variable DoorAddr is assigned to the signal parameter. The transition goes back to the state WaitOpenDoor. All signals, except the signal from the timer DoorTimer, are *saved* in state Wait_DoorTimer.

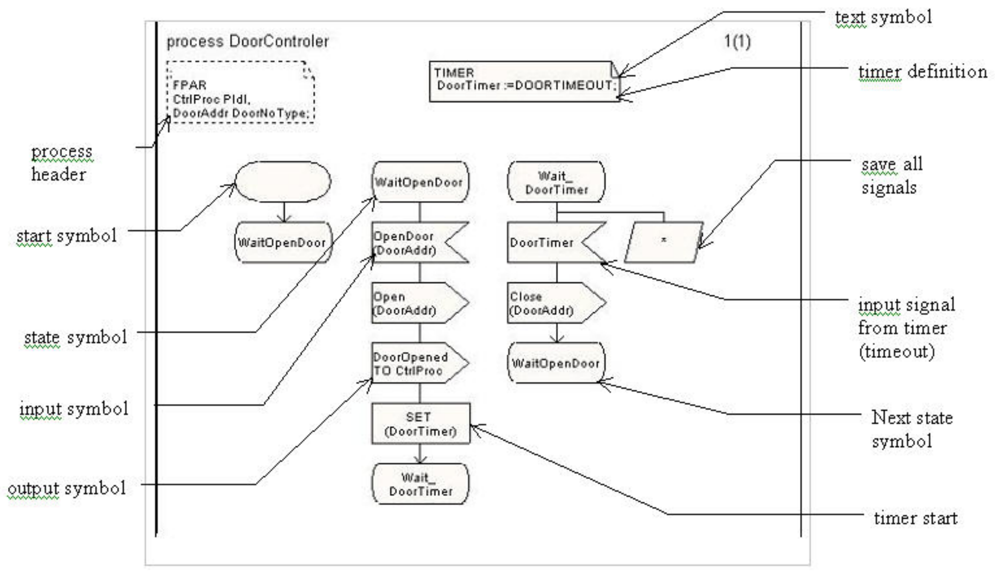


Figure 5: Elements of SDL state machines

SDL state machines have the following semantics. Each SDL process has a so-called *input buffer*. When an SDL process receives a signal, it is first appended to the end of the input buffer. An SDL state machine *consumes* signals from the beginning of the input buffer. When the current state specifies the input of current signal, it is removed from the input buffer and the state machine performs the corresponding *transition*. The signal can be explicitly *saved* in the current state, in which case it remains in the input buffer, and the state machine considers the next signal in the input buffer. When all signals in the input buffer are considered, or when the state machine has finished the current transition, signals in the input buffer are considered again from the beginning of the input buffer. It is possible to save all signals, except those that are explicitly consumed in the current state (the so-called save * statement). When a certain signal is not explicitly consumed or saved, it is consumed implicitly (which means it is deleted from the input buffer and the state machine remains in the current state).

CHAPTER 3

Installing MSC to SDL Synthesizer

In This Chapter

Installing MSC to SDL Synthesizer on a Sun Solaris or HP-UX workstation

To install the KLOCwork MSC to SDL Synthesizer on a UNIX workstation, follow the steps below.

1. Change your current directory to the `solaris` or `hpux` directory on the CDROM, depending on your type of workstation.
2. Run the `msc2sdl-install.sh` script from the directory in your bash shell.
3. The installation program prompts you to enter the path to the directory where you want for the KLOCwork MSC to SDL Synthesizer installed. Enter the path without a slash at the end, for example: `/home/ucs` , then press `Enter`.

Installation begins and all required files are copied to the specified directory. When the installation is completed, follow the instructions for starting KLOCwork MSC to SDL Synthesizer.

Installing MSC to SDL Synthesizer on a PC running Microsoft Windows 98 or Windows NT

To install KLOCwork SC to SDL Synthesizer on your Windows PC, follow the steps below.

1. Open the Windows directory on the KLOCwork MSC to SDL Synthesizer CDROM.
2. Start `setup.exe`
3. Follow the instructions that appear on the screen.

CHAPTER 4

Running the KLOCwork MSC to SDL Synthesizer

In This Chapter

Ways to run the KLOCwork MSC to SDL Synthesizer

There are three ways to run the KLOCwork MSC to SDL Synthesizer:

- through Telelogic Tau
- from the graphical user interface for Microsoft Windows 98 or Microsoft Windows NT
- from the UNIX or Windows command line

All three methods are described in the sections that follow.

Running MSC to SDL Synthesizer from Telelogic Tau

Synthesizer integration with Telelogic Tau

The integration module for KLOCwork MSC to SDL Synthesizer allows you to run it directly from the Telelogic Tau Organizer, using the **MSC to SDL** menu in the Organizer. The Telelogic Tau integration module provides *seamless integration* of the KLOCwork MSC to SDL Synthesizer into Telelogic Tau:

1. Use the Telelogic MSC Editor to create input scenario models in graphical notation. They are automatically converted into MSC textual notation.
2. Apply the KLOCwork MSC to SDL Synthesizer to the selected MSCs. You can place multiple MSCs into an Organizer module. Select one of the two code generators (plain SDL or type-based SDL).
   - Errors and warnings from the MSC to SDL Synthesizer are shown in Organizer Log. The `Show Error` button in Tau Organizer Log automatically opens the MSC Editor at the corresponding symbol of the input MSC model
   - The synthesized SDL model is automatically converted into graphical representation and imported into the Organizer.
   - The synthesized SDL model can be edited using the Telelogic Tau graphical SDL Editor.
   - The SDL model is complete and ready to be analyzed, simulated and validated by the corresponding Telelogic Tau tools.
   - Input MSCs can be directly simulated instead of the synthesized SDL by the Telelogic Tau SDL Simulator (see ***Simulating MSC*** see "Simulating MSCs" on page 27).
   - Synthesis can be controlled by setting some options in the configuration file (see ***Configuration file*** on page 30).

Starting up

To use the KLOCwork MSC to SDL Synthesizer with Telelogic Tau, do the following:

1. Start Telelogic Tau.

2 From the installation directory of KLOCwork MSC to SDL Synthesizer, execute the KLOCwork Tau integration module by typing one of the following commands:

UNIX:

```
> msc2sdl–addin.sh
```

Windows:

```
> msc2sdl–addin.bat
```

With KLOCwork MSC to SDL Synthesizer integrated into Telelogic Tau, a new menu, **MSC to SDL**, appears in the Telelogic Tau Organizer. Use Telelogic Tau to create MSC diagrams, run the Synthesizer to create an SDL model, and continue working with the generated SDL model in Telelogic Tau.

Installation of the KLOCwork MSC to SDL Synthesizer contains scripts sdt.sh (UNIX) and sdt.bat (Windows). This script automatically starts Telelogic Tau and attaches the KLOCwork Tau integration module. You can edit these scripts to point to correct location of Telelogic Tau on your machine.

Selecting MSCs for synthesis

There are two ways to select the input MSCs for SDL synthesis.

- select a single MSC or HMSC diagram in the Telelogic Tau Organizer,
- select a module in the Telelogic Tau Organizer. All MSCs from this module are used as input to the KLOCwork MSC to SDL Synthesizer.

Menus

The **MSC to SDL** menu has the following items:

- Analyze only

  Analyzes selected MSCs. Error and warning messages (if any) are reported to the Organizer Log.
- Synthesize type-based SDL

  Synthesizes a type-based SDL model from selected MSCs. In the case of successful synthesis, SDL GR files are generated and imported into Telelogic Tau. Error and warning messages (if any) are reported in the Organizer Log.

- Synthesize plain SDL

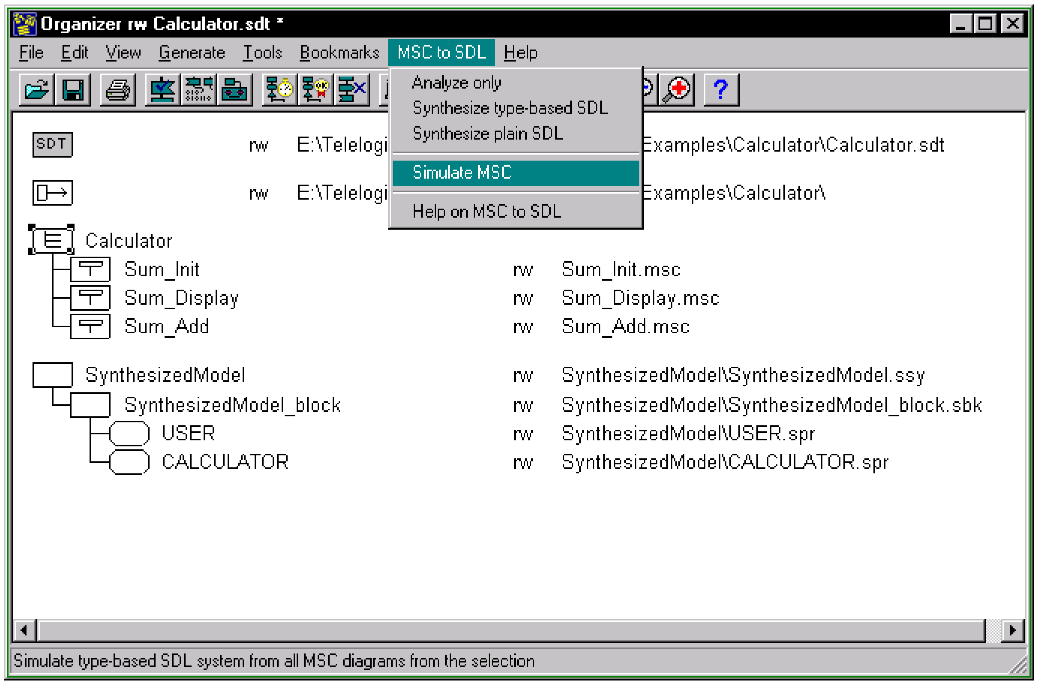
  Synthesizes a plain SDL model from selected MSCs. In the case of successful synthesis, SDL GR files are generated and imported into Telelogic Tau. Error and warning messages (if any) are reported in the Organizer Log.

- Simulate MSC

  Synthesizes a plain SDL model from selected MSCs. In the case of successful synthesis, the synthesized SDL is imported into Telelogic Tau, and analyzed by Telelogic Tau. An executable simulator model is produced and executed using the Telelogic Tau SDL Simulator. The executable model is simulated in terms of the input MSC model. Any error and warning messages are reported in the Organizer Log.

- Help on the KLOCwork MSC to SDL Synthesizer

  Opens an HTML browser and displays the *KLOCwork MSC to SDL Synthesizer Reference Manual*.


Organizer rw Calculator.sdt *
File Edit View Generate Tools Bookmarks MSC to SDL Help
Analyze only
Synthesize type-based SDL
Synthesize plain SDL
Simulate MSC
Help on MSC to SDL
SDT

rw E:\Telelogi Examples\Calculator\Calculator.sdt
rw E:\Telelogi Examples\Calculator\
Calculator
Sum_Init rw Sum_Init.msc
Sum_Display rw Sum_Display.msc
Sum_Add rw Sum_Add.msc
SynthesizedModel rw SynthesizedModel\SynthesizedModel.ssy
SynthesizedModel_block rw SynthesizedModel\SynthesizedModel_block.sbk
USER rw SynthesizedModel\USER.spr
CALCULATOR rw SynthesizedModel\CALCULATOR.spr

Simulate type-based SDL system from all MSC diagrams from the selection

Figure 6: MSC to SDL Synthesizer integrated with Telelogic Tau 4.2

Simulating MSCs

This section describes detailed steps for simulating MSC models using the KLOCwork MSC to SDL Synthesizer and Telelogic Tau. The Tau Integration Module of the KLOCwork MSC to SDL Synthesizer lets you directly simulate the input MSC model instead of the generated SDL. You will be able to

- single step through the MSC model
- guide the simulation of the MSC model
- send application-specific signals from the environment using automatically generated buttons in the Telelogic Tau Simulator
- view the execution trace as symbols on the input MSC are highlighted (Source MSC trace)
- view the execution trace as a new MSC trace generated by the Tau Simulator (MSC trace)

To simulate the MSC models, perform the following steps:

1. Select one or more MSCs in the Telelogic Tau Organizer.
2. Click the Simulate MSC button. The Telelogic Tau Integration Module of the KLOCwork MSC to SDL Synthesizer automatically performs the following sequence of tasks:

   a) Synthesizes plain SDL model from selected MSCs

   b) if synthesis is successful, imports the synthesized SDL into Telelogic Tau

   c) analyzes the synthesized model using the SDL Analyzer of Telelogic Tau

   d) generates C code for the synthesized SDL model using the Tau SDL Code generator

   Note: This step requires that the Telelogic Tau Code generator be installed.

   e) calls the C compiler to produce the executable simulator corresponding to the synthesized SDL model.

   Note: This step requires a C compiler to be available.

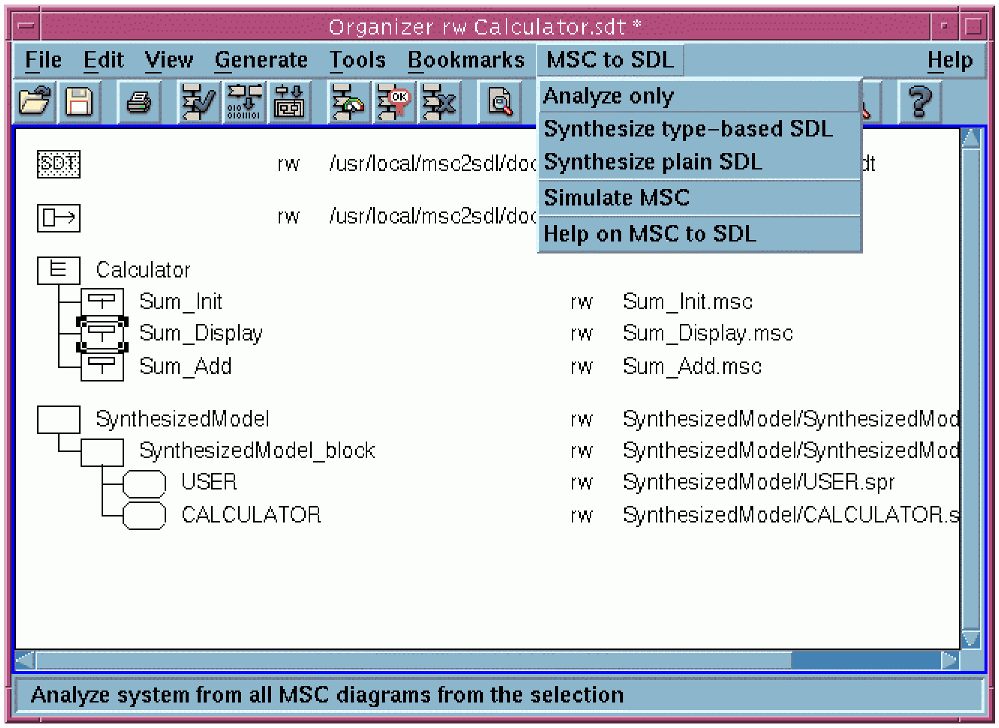
   f) automatically generates the button definitions for the Simulator GUI. The generated buttons correspond to the signals that can be sent from the environment of the synthesized SDL model.

g) if generation in steps c to f above is successful, starts the Simulator GUI with the automatically generated buttons (see ***SDL Simulator with application-specific buttons*** "Figure 8: SDL Simulator with application specific buttons" on page 29)

If any problems are encountered, error and warning messages are reported in the Telelogic Tau Organizer Log. In particular, errors and warnings may be produced during the following steps:

- MSC to SDL Synthesizer (syntax and semantic errors in the input MSC)
- SDL Analyzer (syntax and semantic errors in data operations)
- C environment errors (environment is incorrectly installed)

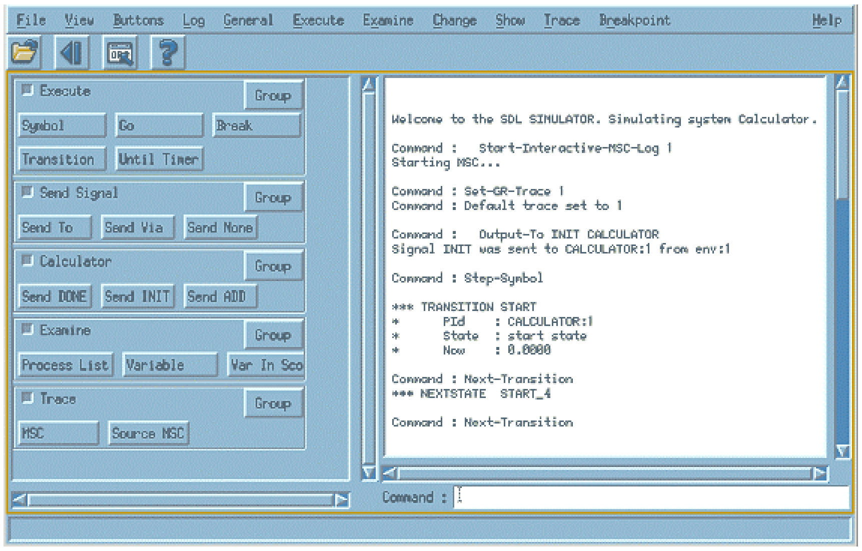
Figure 7: MSC to SDL menu inTelelogic Tau Organizer



rw /usr/local/msc2sdl/doc
rw /usr/local/msc2sdl/doc
rw Sum_Init.msc
rw Sum_Display.msc
rw Sum_Add.msc
rw SynthesizedModel/SynthesizedMod
rw SynthesizedModel/SynthesizedMod
rw SynthesizedModel/USER.spr
rw SynthesizedModel/CALCULATOR.s

Analyze system from all MSC diagrams from the selection

Figure 8: SDL Simulator with application specific buttons

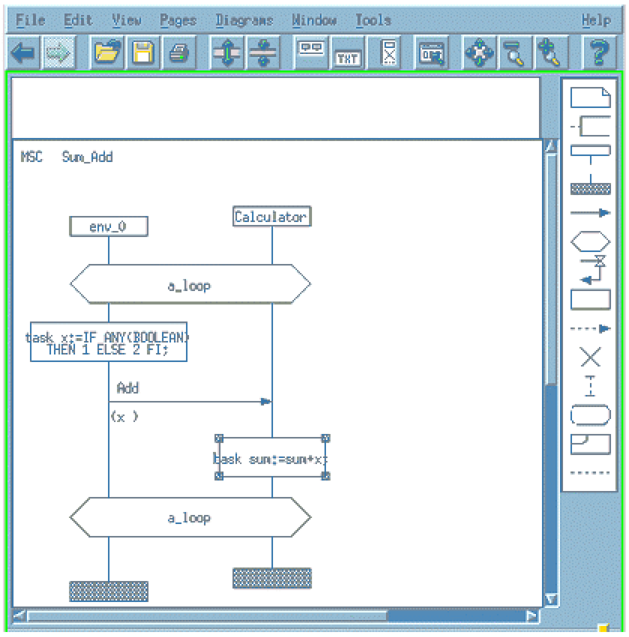


```
Welcome to the SDL SIMULATOR. Simulating system Calculator.

Command :   Start-Interactive-MSC-Log 1
Starting MSC...

Command : Set-GR-Trace 1
Command : Default trace set to 1

Command :   Output-To INIT CALCULATOR
Signal INIT was sent to CALCULATOR:1 from env:1

Command : Step-Symbol

*** TRANSITION START
*       PId     : CALCULATOR:1
*       State   : start state
*       Now     : 0.0000

Command : Next-Transition
*** NEXTSTATE  START_4

Command : Next-Transition
```

Once the Simulator is started, you can explore the behavior of the input MSC model using the capabilities of the SDL Simulator:

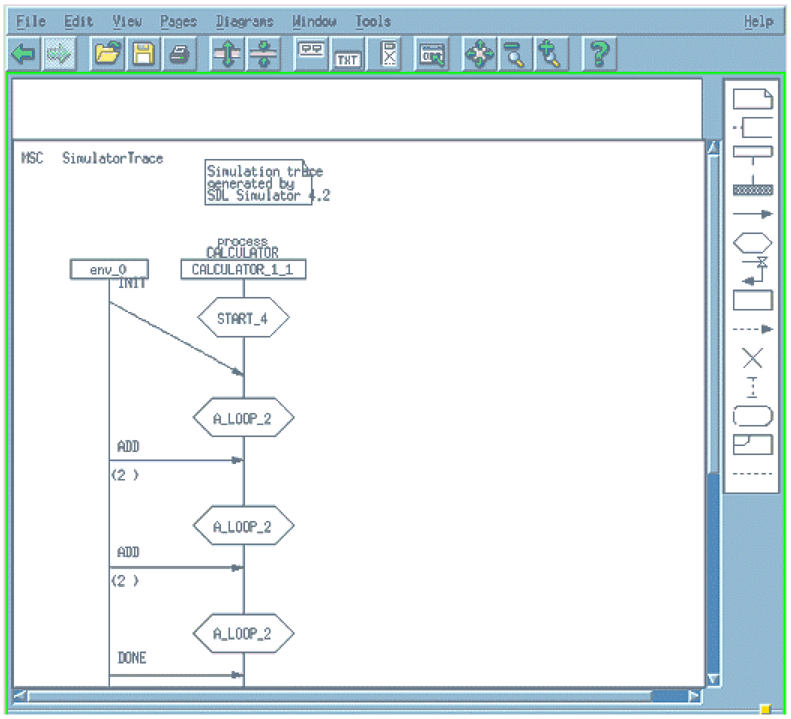
- Use the Symbol button in the Execute group to single-step through your MSC model. The right hand panel displays the textual trace of events generated by the synthesized SDL model.
- Use the Source MSC button in the Trace group to highlight events directly in the input MSC diagrams (as shown in the following figure).

Figure 9: Showing trace as source MSC in MSC Editor of Telelogic



- Use the MSC button in the Trace group to see the events as the MSC trace, generated by the SDL Simulator (as shown in the following figure). Use the Trace pull-down menu to set the desired granularity of the generated trace.

Figure 10: Showing MSC trace generated by the SDL Simulator



- Use other buttons in the Execute group to control the execution of the model (refer to Telelogic Simulator manual).
- Use buttons in the application-specific group (for example, this group is named Calculator in ***SDL Simulator with application specific buttons*** "Figure 8: SDL Simulator with application specific buttons" on page 29) to send specific signals from the environment of the model (refer to Telelogic Simulator manual).
- From the Organizer File menu, chose Restart to restart simulation.

Configuration file

To set the options for the KLOCwork MSC to SDL Synthesizer when it is integrated with Telelogic Tau, edit the configuration file `msc2sdl.ini`.

The Synthesizer searches for msc2sdl.ini file in the following sequence:

1. Telelogic Tau model directory
2. Current working directory (the one from which Telelogic Tau was started)
3. Installation directory

The contents of the configuration file have the following syntax:

```
<file> ::= <assignment> *

<assignment> ::= <option> = <value>
```

Each <assignment> sentence should be written on a separate line. Command options are listed below. For each option, the set of possible values is provided.

Option	Value	Description
`OPT_UPPERCASE`	yes, no	Performs a case sensitivity analysis of identifiers. By default (=yes), all identifiers are automatically transformed to upper case.
`OPT_ALL_WARNINGS`	yes, no	Displays both critical and non-critical warning messages. By default (=no), only critical messages are produced.
`OPT_CROSS_REFS`	yes, on	Generates references to the source MSCs in the generated SDL. They are generated as comments and can be used by Telelogic Tau to simulate MSCs (see ***Simulating MSCs*** on page 27).
`GENERATE_SYSTEM_NAME`	"<name>"	Specifies the name of the synthesized system. It is also the prefix for some other identifiers in the generated SDL files. The default is `'SynthesizedModel'`.
`OPT_EXPAND_COREGIONS`	yes, no	Generates permutations for active events in coregions (see ***Synthesis: Coregions*** on page 61).
`SLICE`	"<inst1>…<instn >"	Selects a subset of instances to be included into the generated SDL system. Includes <instance> to the slice (see ***Slices*** "Synthesis: Slices of MSCs" on page 81). Multiple instances can be included into the slice. When this option is not used, then all instances are included into the generated SDL system.

Option	Value	Description
`OPT_END_HANDLING`	state, stop, merge	Sets the way of handling the control flow end. Possible values of type are: state (default), stop, merge (see ***Control flow end*** "Synthesis: End of instance handling" on page 77).
OPT_CLEAN_GEN_PR	yes, no	Cleans generated phrase representation (PR) files from the temporary directory. By default (value=no), all PR files are left in the directory GEN_PR.

Note: Whitespace, empty lines and lines starting with "#" or ";" characters are ignored.

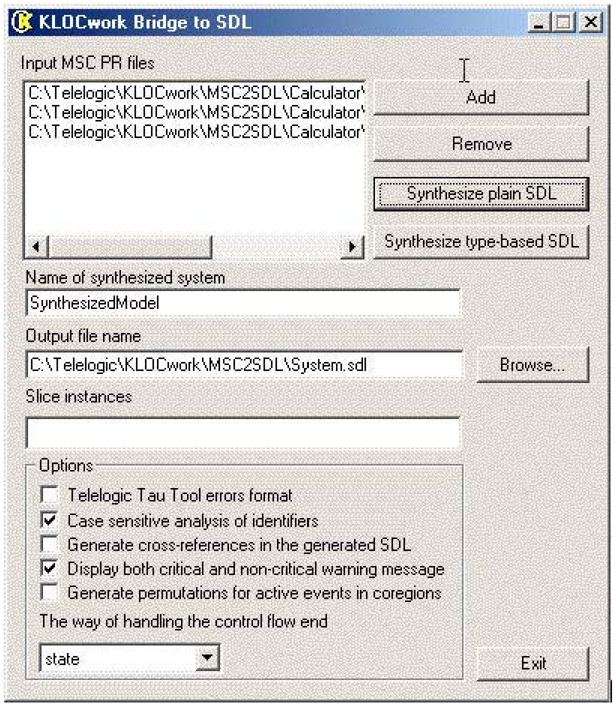
Running from the GUI for Windows 98/NT

You can run the KLOCwork MSC to SDL Synthesizer in stand-alone mode from a graphical user interface (GUI) for Microsoft Windows 98 or Microsoft Windows NT.

Note: MSC textual representation (see ***MSC PR syntax*** on page 99) should be used as the input to the Synthesizer in stand-alone mode.

To start the GUI, click on the MSC2SDL icon in the KLOCwork MSC to SDL Synthesizer installation directory. You can create the shortcut for this application.

The KLOCwork MSC to SDL Synthesizer GUI lets you select MSC PR files in the multi-file selection dialog, which is started by pressing the Add button. To remove MSC PR files from the list, select them and press the Remove button.



Input MSC PR files

C:\Telelogic\KLOCwork\MSC2SDL\Calculator\
C:\Telelogic\KLOCwork\MSC2SDL\Calculator\
C:\Telelogic\KLOCwork\MSC2SDL\Calculator\

Name of synthesized system: SynthesizedModel

Output file name: C:\Telelogic\KLOCwork\MSC2SDL\System.sdl

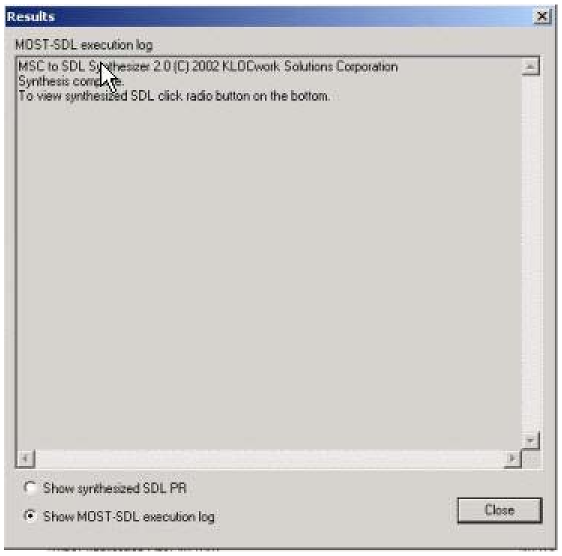
Slice instances:

Options

- ☐ Telelogic Tau Tool errors format
- ☑ Case sensitive analysis of identifiers
- ☐ Generate cross-references in the generated SDL
- ☑ Display both critical and non-critical warning message
- ☐ Generate permutations for active events in coregions

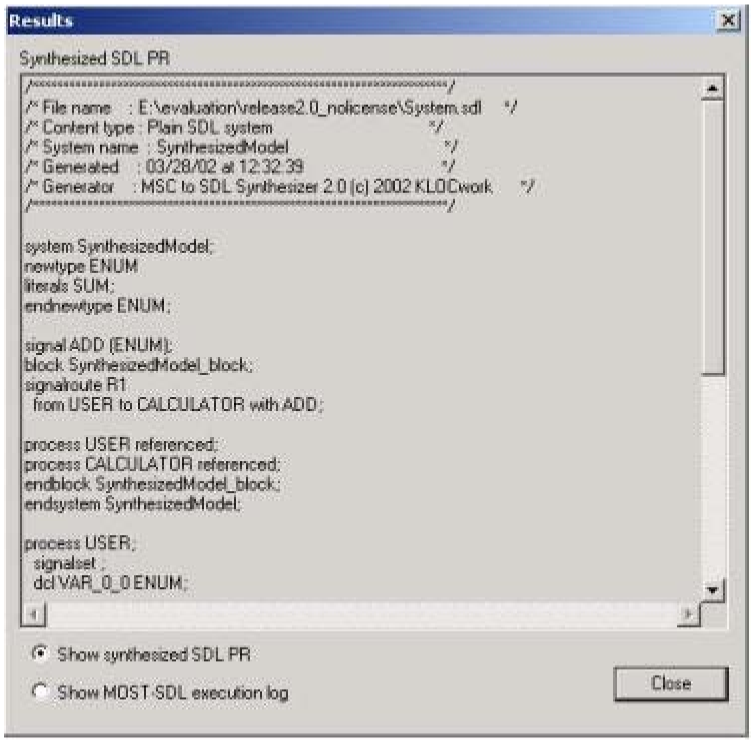
The way of handling the control flow end: state

The KLOCwork MSC to SDL Synthesizer GUI displays the panel with the log.



```
MSC to SDL Synthesizer 2.0 (C) 2002 KLOCwork Solutions Corporation
Synthesis complete.
To view synthesized SDL click radio button on the bottom.
```

Click the Show synthesized SDL PR radio button to view resulting SDL in phrase representation.



```
/*************************************************************************/
/* File name    : E:\evaluation\release2.0_nolicense\System.sdl    */
/* Content type : Plain SDL system                                 */
/* System name  : SynthesizedModel                                  */
/* Generated    : 03/28/02 at 12:32:39                              */
/* Generator    : MSC to SDL Synthesizer 2.0 (c) 2002 KLOCwork      */
/*************************************************************************/

system SynthesizedModel;
newtype ENUM
literals SUM;
endnewtype ENUM;

signal ADD (ENUM);
block SynthesizedModel_block;
signalroute R1
 from USER to CALCULATOR with ADD;

process USER referenced;
process CALCULATOR referenced;
endblock SynthesizedModel_block;
endsystem SynthesizedModel;

process USER;
 signalset ;
 dcl VAR_0_0 ENUM;
```

The following check buttons define parameters of the synthesis algorithm transferred to the KLOCwork MSC to SDL Synthesizer through the command line. (Equivalent configuration file options are given in brackets.)

- Generate error messages in Telelogic Tau error format (OPT_SDTREF)
- Perform case-sensitive analysis of identifiers (OPT_UPPERCASE)
- Generate cross-references in the generated SDL (OPT_CROSS_REFS)
- Display both critical and non-critical warning messages (OPT_ALL_WARNINGS)
- Generate permutations for active events in coregions (For more information see, ***Synthesis: Coregions*** on page 61.)

In the *Slice instances* field, you can enter as list of slice instances (separated by spaces). See ***Synthesis: Slices of MSCs*** (on page 81).

Press Synthesize plain SDL or Synthesize type-based SDL to start the analysis. The synthesized SDL (in text format is placed in the file with the name entered in the *Output file name* field.

Running the KLOCwork MSC to SDL Synthesizer from the command line

You can run the KLOCwork MSC to SDL Synthesizer in stand-alone mode from the UNIX or Windows command line.

Note: MSC textual representation (see ***MSC PR syntax*** on page 99) should be used as the input to the Synthesizer in stand-alone mode.

Use one of the following programs to run the KLOCwork MSC to SDL Synthesizer from the command line. These programs are located in the KLOCwork MSC to SDL Synthesizer installation directory.

- `tbsdl` – synthesize type-based SDL
- `plainsdl` – synthesize plain SDL
- `msc2sdl` – synthesize plain or type-based SDL depending on command line options

On Windows, these programs are batch files.

The command-line options for `msc2sdl` are described below. The programs `tbsdl` and `plainsdl` use the same set of options except for the options –t.

`msc2sdl` has a simple command-line interface. The input to the Synthesizer is a set of text files in extended MSC-PR language (for more information, see ***MSC PR syntax*** on page 99). The output of the KLOCwork MSC to SDL Synthesizer is a simple text file containing the synthesized SDL model.

```
msc2sdl [ options ] <input-file>s ...
```

The following options are available:

`-s <system-name>`
`--system <system-name>`
	Specify the name of the synthesized system. It is also the prefix for some other identifiers in the generated SDL files. The default is `SynthesizedModel`.

`-o <output-file>`
`--output <output-file>`
	Specify the output file for the generated SDL specification. The default is `SynthesizedModel.sdl`

`-t <codegen>`
`--codegen <codegen>`
	Specify the SDL code generator for the msc2sdl program. The available code generators are:
	▪ `plainsdl` – for plain SDL;
	▪ `tbsdl` – for type-based SDL.
	▪ If this option is missing, no SDL is generated but error and warning messages are reported for the input MSCs.

`-r` `--sdtref`	Generate error messages (and cross-references, if needed) compatible with the Telelogic Tau format. For the description of error messages see ***Error messages*** 107
`-c` `--case`	Perform a case sensitivity analysis of identifiers. By default, all identifiers are automatically transformed to upper case.
`-x` `--xref`	Generate references to the source MSCs in the generated SDL. They are generated as comments and can be used by Telelogic Tau to simulate MSCs (see ***Simulating MSC*** see "Simulating MSCs" on page 27)
`-w` `--all-warn`	Display both critical and non-critical warning messages. By default only critical messages are produced.

`-S <instance>`
`--slice <instance>`
	Include <instance> to the slice (see ***Slices*** "Synthesis: Slices of MSCs" on page 81). Multiple –S options may be used to specify a slice consisting of several instances. If no –S options are given no slice is selected (this is equivalent to a slice consisting of all instances).

`-p` `--expand_coregions`	Generate permutations for active events in coregions (see ***Synthesis: Coregions*** on page 61).

`-e type`
`--end type`
	Set the way of handling the control flow end. Possible values of type are: state (default), stop, merge (see ***Control flow end*** "Synthesis: End of instance handling" on page 77).

`<input-file>`	Specify a text file containing the description of one or several diagrams in extended MSC language (for description of the textual representation of MSCs see ***MSC PR syntax*** on page 99).

CHAPTER 5

Synthesis: Basic MSC scenarios

In This Chapter

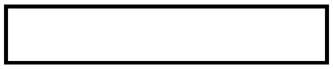
Introduction

Synthesis is simplest when an input model consists of a single MSC without conditions and coregions (basic MSC). Such an MSC defines the behavior of the set of actors. A basic MSC diagram describes a (single) sequence of events along the instance axis of each actor. More complex behavior of actors, including alternative behaviors and repetitive behavior can be described using compositions of basic MSCs (see ***Synthesis: Compositions of multiple MSCs*** on page 63).

Diagram

A Message Sequence Chart (MSC) diagram describes the message flow between *instances*. One MSC describes a partial behavior of a system.

```
<msc diagram> ::=     <simple msc diagram> | <hmsc diagram>

<simple msc diagram> ::=   <msc symbol> contains

         {<msc heading> <msc body area> }
<msc symbol> ::=   <frame symbol>
            is attached to { <external message area> * } set
<frame symbol> ::=
```



```
<msc heading> ::=  msc <msc name>
<msc body area> ::= { <instance layer>
                     | <text layer>
                     | <event layer>
                     | <connector layer> } set

<event layer> ::= <event area> |< event area>

            above <event area>
<instance layer> ::= {<instance area> * } set
<text layer> ::= {<text area> * } set

<event area> ::= <instance event area>

            | <shared event area>
            | <create area>

<instance event area> ::= <message event area>

                  | <timer area>
                  | <concurrent area>
                  | <action area>
<connector layer> ::= { <message area> *
                       | <incomplete message area> * } set
<shared event area> ::= <condition area>
```

An MSC diagram is mapped onto an SDL system containing a single SDL block (see detailed illustration in the next section). Plain SDL mapping uses a textual definition of the SDL system, which contains the textual definition of the block. The generated SDL block, in turn, contains other textual definitions, corresponding to MSC instances and data. These definitions cannot be reused, except using the cut-and-paste approach. The type-based SDL mapping produces an SDL package that contains a collection of the so-called SDL structured types for the generated SDL block as well as for other definitions. The generated SDL system uses the generated SDL package and contains the type-based instance of the system, which simply uses the corresponding generated block type from the package. This allows reuse of the generated SDL definitions to, for example, instantiate them in a different context.

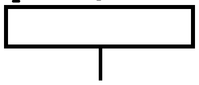
Instance

An MSC is composed of interacting *instances*. An instance of an instance *kind* has the properties of this kind. Within the instance *heading* the instance kind name may be specified in addition to the instance *name*.

An *Instance head* symbol determines the start of a description of the instance within an MSC. It does not describe the creation of the instance. Correspondingly, the *instance end* symbol determines the end of a description of the instance within an MSC. It does not describe the termination of the instance. All instance fragments with the same name constitute the same instance.

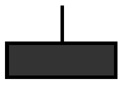
The instance definition provides an event description for message inputs and message outputs, actions, shared and local conditions, timer events, instance creation and instance stop. Within the instance body the ordering of events is specified. Outside of *coregions* a total ordering of events is assumed. Within coregions no ordering of events is assumed. For a detailed description of coregions, see ***Synthesis: Coregions*** on page 61 .

```
<instance area> ::=  <instance fragment area>
                     [is followed by <instance body area>]
<instance fragment area> ::=  <instance head area>
                      is followed by <instance body area>
<instance head area> ::= <instance head symbol>
                      is associated with <instance heading>
                     [is attached to <createline symbol> ]
<instance heading> ::= <instance name
                     [:<instance kind> ]
                        [decomposed]
<instance head symbol> ::=
```



```
<instance name>  ::= <name>
<instance kind>  ::= [ <kind denominator> ] <kind name>
<kind denominator>  ::= system | block | process
<kind name>   ::= <name>
<instance body area>::= <instance axis symbol>
              is attached to { <event area> * } set
              is followed by { <instance end symbol> |
                        <stop symbol> }
<instance axis symbol> ::=
```

```
<instance end symbol> ::=
```



MSC instances are mapped onto SDL processes (plain SDL) or process types (type-based SDL). Both mappings are illustrated below.

Figure 11: Plain SDL MSC instance mapping (SDL GR)

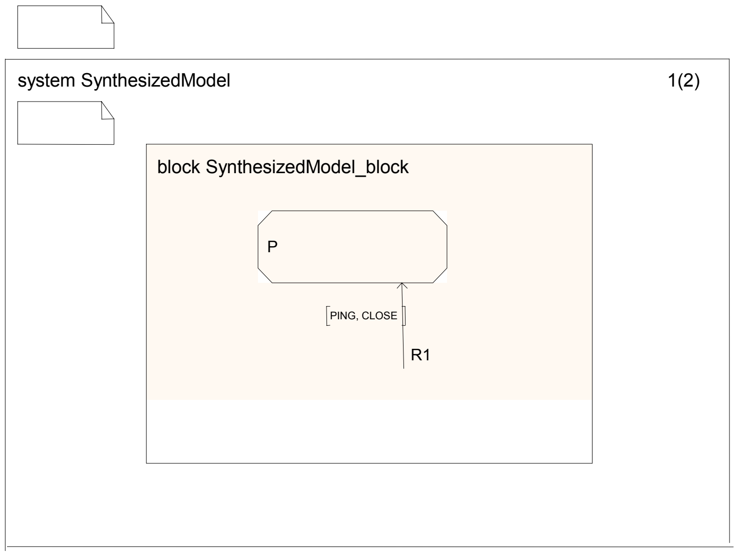
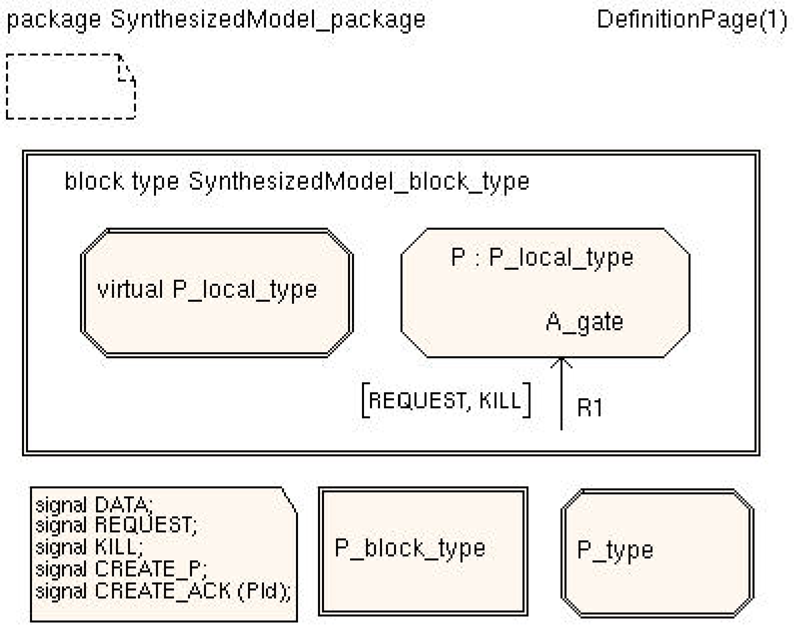


Figure 12: Type-based SDL MSC instance mapping



Message

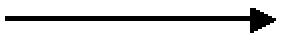
A message within an MSC is a relation between an *output* and an *input*. For an MSC, the message output denotes the message sending, the message input denotes the message consumption. No special construct is provided for message reception (input into the buffer). The output may come from either the *environment* or an *instance*, or be *found*; and an input is to either the *environment* or an *instance* or is *lost*. An *incomplete message* is a message, which is either an output (where the input is lost/unknown) or an input (where the output is found/unknown).

A message exchanged between two instances can be split into two *events*: the message input and the message output. In a message *parameters* may be assigned.

The correspondence between message outputs and message inputs has to be defined uniquely. In a graphical representation, a message is represented by an *arrow*.

The loss of a message, for example a case in which a message is sent but not consumed, is identified by a *black hole* in the graphical representation. Symmetrically, a spontaneously found message (a message that appears from nowhere) is defined by a *white hole* in the graphical representation.

```
<message event area> ::= { <message out area>
                           | <message in area> }
<message out area> ::= <message out symbol>
                  is attached to  <instance axis symbol>
                  is attached to <message symbol>
<message out symbol> ::= <void symbol>
<void symbol> ::=  .
<message in area> ::=   <message in symbol>
                  is attached to <instance axis symbol>
                  is attached to <message symbol>
<message in symbol> ::= <void symbol>
<message area> ::=      <message symbol>
                  is associated with <msg identification>
                  is attached to {<message start area>
 |<message end area> }
<message start area> ::= <message out area>

<message end area> ::= <message in area>

<message symbol> ::=
```

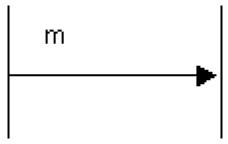


```
  <incomplete message area> ::= { <lost message area>
                  | <found message area> }
```

```
<lost message area> ::= <lost message symbol>
                is associated with <msg identification>
                [is associated with <instance name> ]
                is attached to <message start area>
<lost message symbol> ::= ──►●
<found message area> ::= <found message symbol>
                is associated with <msg identification>
                [is associated with <instance name> ]
                is attached to <message end area>
<found message          ◄──○
symbol> ::=
<external message area> ::= { <external in message area>
                       | <external out message area> }
                is attached to <msc symbol>
<external in message area> ::= <void symbol>
                is attached to { <message symbol>
                      | <found message symbol> }
<external out message symbol> ::= <void symbol>
                is attached to { <message symbol>
                       | <lost message symbol> }
<msg identification> ::= <message name>
           [,<message instance name>] [(<parameter list>)]
<parameter list> ::= <parameter name>[,<parameter list>]
<parameter name> ::= <name>
<message name> ::= <name>
<message instance name> ::= <name>
```

A simple message exchange without parameters is illustrated as follows:

Figure 13: Message output

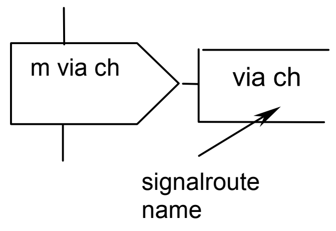


where m is the message name

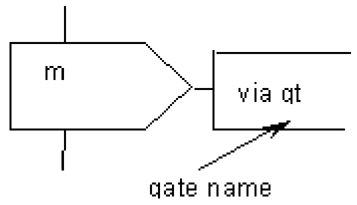
Mapping of message output

A message output event without parameters is mapped to the following SDL constructs in the sender SDL process.

Plain SDL

Phrase representation	Graphical representation
... output m via ch; ...	

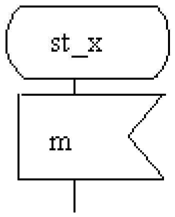
Type-based SDL

Phrase representation	Graphical representation
... output m via gt; ...	

Mapping of message input

Message input event without parameters to the following SDL constructs in the receiver SDL process (GR and PR representations are identical for plain and type-based SDL):

Phrase representation	Graphical representation

...
state st_x;
input m;
...
	

Additionally, the corresponding SDL *signal definition* is generated according to the rules described in ***Synthesis: Data manipulations in MSCs*** (on page 83) .

In MSC diagrams, instances with names *env_0*, *environment* and *env* represent the environment of the system. That is,

- a message output to env_0, environment or env is interpreted as output to the environment
- a message input from env_0, environment or env is treated similarly
- events on axis env_0, environment or env, other than message inputs and outputs, are ignored

When the MSC slice is specified, instances other than those included in the slice are also treated as the environment of the systems (see ***Synthesis: Slices of MSCs*** (on page 81)).

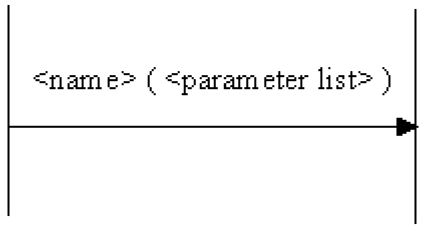
The KLOCwork MSC to SDL Synthesizer assumes the following semantics for *lost* and *found* messages:

- a lost message is interpreted as sent to the environment
- a found message is interpreted as received from the environment

The KLOCwork MSC to SDL Synthesizer handles MSC messages with parameters. In SDL, the sender provides a signal parameter value. The receiver stores this value to some variable. The KLOCwork MSC to SDL Synthesizer lets you specify both the value and the variable in message parameters on the MSC diagram.

The syntax of parameters is as follows:

Figure 14: Message with parameters



```
<parameter list> ::= <parameter> {, <parameter> }*
<parameter> ::=
        <variable name>
      | <variable name> := <expr>
      | <expr> =: <variable name>
      | <constant>
```

<expr> is any text with balanced brackets. The KLOCwork MSC to SDL Synthesizer does not treat quotes as string delimiters when parsing message parameters.

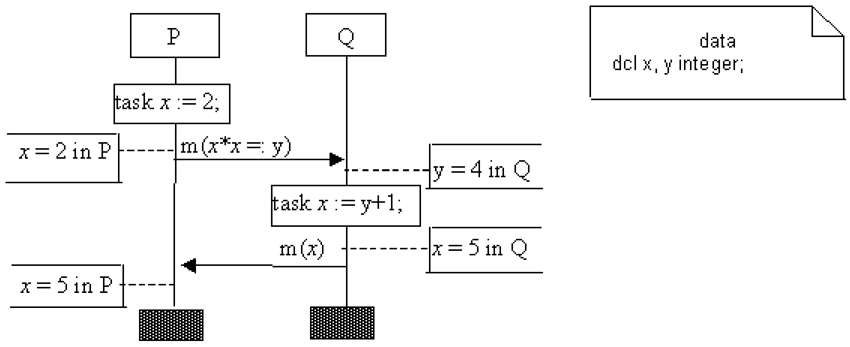
If a message parameter has the form *x* := *e* (where *x* is a variable and *e* is an expression), then

- the sender evaluates *e* and passes it as the signal parameter
- receiver stores the received parameter value to variable *x*

Form *e* =: *x* is equivalent to *x* := *e*. If a parameter is just one variable name *x,* it is equivalent to *x* := *x* (that is, the value of *x* is copied from the sender to the receiver).

Semantics of messages with parameters are further illustrated in the following figure. Data definitions are defined in ***Synthesis: Data manipulations in MSCs*** (on page 83). Note that all variables are local to actors, therefore, each actor has its own set of variables with different values. Exchange of messages with parameters synchronizes the values of variable in different actors.

Figure 15: Semantics of messages with parameters



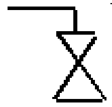
Timer

In MSCs, either the setting of a *timer* and a subsequent *time-out* due to timer expiration or the *setting* of a timer and a subsequent timer *stop* may be specified. In addition, the individual timer constructs – timer setting, stop/time-out - may be used separately, for example in a case in which timer expiration or time supervision is split between different MSCs. In the graphical representation, the start symbol has an *hourglass* form connected to the instance axis by a line symbol. A message arrow pointing at the instance, which is attached to the hourglass symbol, describes time-out. The timer stop symbol has the form of a *cross* that is connected with the instance by a *line*.

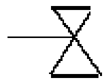
The specification of *timer instance name* and *timer duration* is optional.

Start denotes setting the timer and stop denotes canceling of the timer. Time-out corresponds to the consumption of the timer signal.

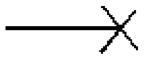
```
<timer area>::= <timer start area>|<timer stop area>|<timeout area>
<timer start area> ::= <timer start symbol>
                 is associated with <timer name>
                 [<duration>]
                 is attached to <instance axis symbol>
                 [ is attached to { <timer stop symbol2>
                  | <timeout symbol3> } ]
<timer start symbol> ::= <start symbol1> | <start symbol2>
<start symbol1> ::=
```



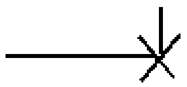
```
<start symbol2> ::=
```



```
<timer stop area> ::= <timer stop area1>
                    | <timer stop area2>
<timer stop area1> ::= <timer stop symbol1>
               is associated with <timer name>
               is attached to <instance axis symbol>
<timer stop area2> ::= <timer stop symbol2>
               is associated with <timer name>
                is attached to <instance axis symbol>
               is attached to <timer start symbol>
<timer stop symbol1> ::=
```

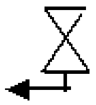


```
<timer stop symbol2> ::=
```

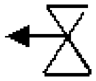


```
<timeout area> ::= <timeout area1> | <timeout area2>
```

```
 <timeout area1> ::= <timeout symbol>
               is associated with <timer name>
               is attached to <instance axis symbol>
<timeout symbol> ::= <timeout symbol1> | <timeout symbol2>
<timeout symbol1>
::=
```



```
<timeout symbol2>
::=
```



```
<timeout area2> ::= <timeout symbol3>
              [is associated with <timer name>]
               is attached to <instance axis symbol>
               is attached to <timer start symbol>
<duration name> ::= <expr>
<timer name> ::= <name>
```

An optional duration value in a timer set event is interpreted as a value of time interval. Time is measured between the timer set event and the timer reset or timeout event connected to it. Set, reset and timeout events might be split or might be connected by a line.

The duration value may be an expression (in particular a constant).

For example, two timer events (illustrated in the following figures) are equivalent.

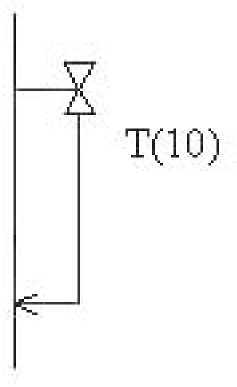
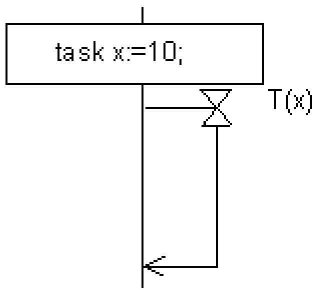


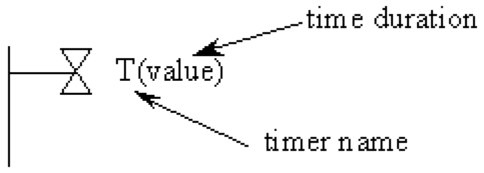
Figure 16: Timer duration can be a constant

Figure 17: Timer duration can be an expression

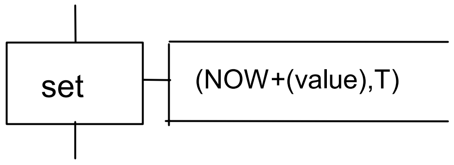


The timer start, stop and timeout events are mapped to SDL as follows:

Timer start event (shown below) is mapped to the same representation for plain and type-based SDL (see the figures that follow).

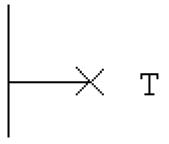


Mapping of timer start event (Plain or type-based SDL)

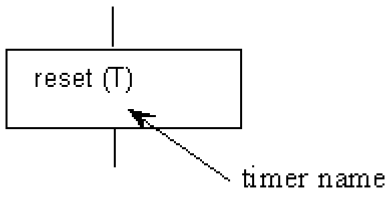
Phrase representation	Graphical representation
... set (NOW + (value),T); ...	

Timer stop event is mapped to the same construct in plain and type-based SDL (as shown in the following figures).

Figure 18: Timer stop event

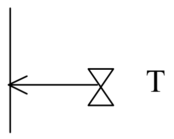


Mapping of timer stop event (Plain or type-based SDL)

Phrase representation	Graphical representation
... reset(T); ...	

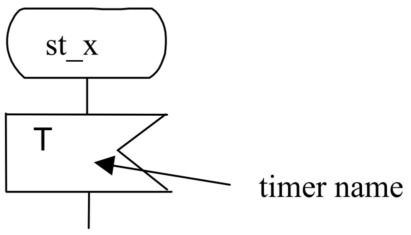
Timeout event is mapped to the message input where the name of which is the same as the timer name for both plain and type-based SDL (as shown in the following figures).

Figure 19: Timeout event



Mapping of timeout event (Plain or type-based SDL)

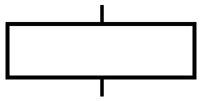
Phrase representation	Graphical representation

…
state st_x;
input T;
…
	

Action

In addition to message exchange, the *actions* may be specified in MSCs. An action is an atomic event that has formal *data statements* associated with it.

```
<action area> ::= <action symbol>
             is attached to <instance axis symbol>
             contains <action statement>
<action symbol> ::=
```



```
<action statement> ::= <text>
```

Any valid SDL statement may be specified in an action statement. This action statement is not analyzed by the KLOCwork MSC to SDL Synthesizer and is transferred directly to the generated SDL file. In particular, one can use SDL `task` (see figure "Figure 20: Assignment task statements in action statements" on page 53) or `call` (see figure "Figure 21: Call statement in action statements" on page 53) statements in action statements.

Figure 20: Assignment task statements in action statements

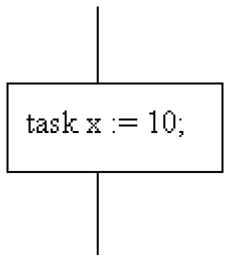
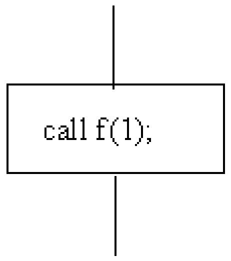
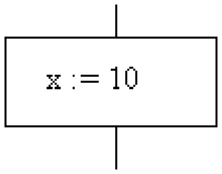


Figure 21: Call statement in action statements



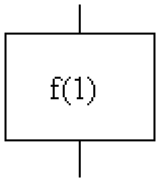
SDL procedures used in `call` statements may be defined for example after `process` keyword in a text symbol. The task statement is mapped to the following ***SDL statement*** (on page 54).

Plain or type-based mapping of a task statement

Phrase representation	Graphical representation
... task x:=10; ...	

The call statement is mapped to the following ***SDL statement*** (on page 54).

Plain or type-based mapping of a call statement

Phrase representation	Graphical representation
... call f(1); ...	

For more details of using SDL data in the input MSC models, see ***Synthesis: Data manipulations in MSCs*** (on page 83).

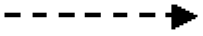
Instance creation

Creation and *termination* of instances can be specified within MSCs. An instance may be created by another instance. No message events before the creation can be attached to the created instance.

The *creation* event is depicted by the end of the <createline symbol> that has no arrowhead. The creation event is attached to the instance axis. If the <create area> is generally ordered, this ordering applies to the creation event. The arrowhead points to the <instance head symbol> of the created instance.

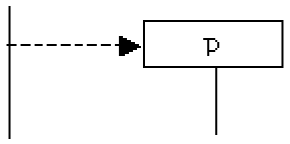
Create defines the dynamic creation of an instance by another. Dynamically there can be only one creation in the life of an instance and no events on the instance may take place before its creation.

```
<create area> ::= <createline symbol>
           [is associated with <parameter list> ]
           is attached to <instance axis symbol>
<instance head area>

<createline symbol> ::=
```

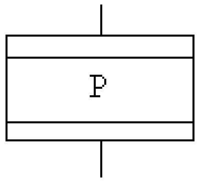


In general, the MSC create event is mapped to the following SDL constructs for the SDL process-creator in the plain and type-based SDL mappings of the create event (as shown in the following figures).

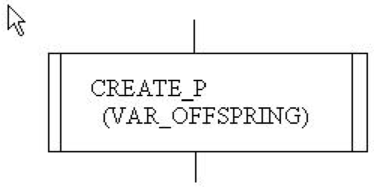
Figure 22: Create event without parameters



Mapping of create event, Plain SDL

Phrase representation	Graphical representation
… create P; …	

Mapping of create event, Type-based SDL

Phrase representation	Graphical representation
... call CREATE_P (VAR_OFFSPRING); ...	

In the type-based SDL model (shown in the following figure), process creation is implemented as follows:

- The generated block type has a special "manager" process for creating other processes. This "manager" process receives the create message and creates the required process. See ***"Manager" process P_gen*** (see "Figure 23: "Manager" process P_gen" on page 57). With the CREATE_ACK message, it returns the new process PID.
- The parent process has procedure CREATE_P for each child process P it creates. See ***Procedure CREATE_P*** (see "Figure 24: Procedure CREATE_P" on page 57). This procedure performs all necessary interprocess communications process creation. It returns the PID of the created process. By calling this procedure, this PID is stored to the variable VAR_OFFSPRING of the parent process.

Type-based SDL, Graphical representations

Figure 23: "Manager" process P_gen

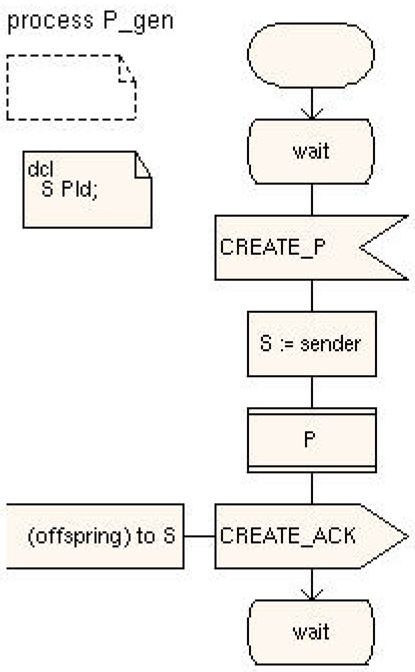
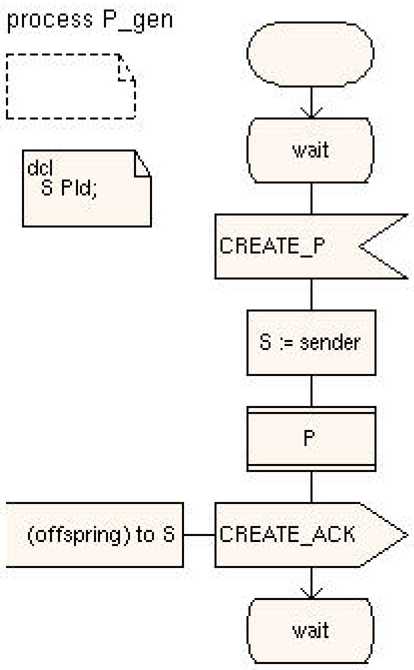


Figure 24: Procedure CREATE_P



Create parameters and message parameters are handled similarly to message parameters. The created process receives values through its formal parameters and may analyze them and assign to local variables.

The following syntax is used:

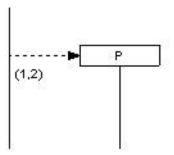


Figure 25: Create event with parameters

Instance stop

The instance *stop* is the counterpart to the instance creation, except that an instance can only stop itself whereas an instance is created by another instance.

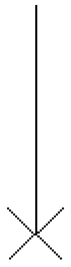
The top at the end of an instance represents the *termination* of this instance. Dynamically there can be only one stop event in the life of an instance and no events may take place after the instance stop.

`<stop symbol> ::=` **X**

The KLOCwork MSC to SDL Synthesizer interprets `endinstance` and `stop` symbols in the following way:

- the `stop` symbol denotes a termination of the corresponding instance
- the `endinstance` symbol means that the behavior of the instance is not defined after this symbol on the given MSC

Instance stop

Phrase representation	Graphical representation
.... stop;	

CHAPTER 6

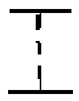
Synthesis: Coregions

Coregion makes it possible to describe areas where events may come in any order. Such a coregion in particular covers the practically important case of two or more incoming messages where the ordering of consumption may be interchanged. Conversely, when broadcasting messages, the order of two or more outgoing messages may be interchanged.

For MSCs a total ordering of events is assumed within each instance. By means of a coregion an exception to this can be made: events contained in the coregion are unordered.

If a timer start and the corresponding time-out or stop are contained in a coregion, then an implicit general ordering is assumed between the start and the time-out/stop.

```
<concurrent area> ::= <coregion symbol>
       is attached to <instance axis symbol>
       contains <coevent area>
<coregion symbol> ::=
```



```
<coevent area> ::=  { <message event area>
                            | <incomplete message area>
                            | <action area>
                            | <timer area>
                            | <create area> } *
```

In the generated system all these events are executed sequentially. The order of their execution depends on the following:

- events occurring during the runtime (like signal passing)
- decisions made by the KLOCwork MSC to SDL Synthesizer synthesis algorithms

Note that normally the fixed order of event execution does not restrict the functionality of the generated SDL process. This is achieved due to `save *` constructs in SDL states. For instance, a coregion contains inputs of messages a and b. The Synthesizer may generate their input in some fixed order, for example, first a and then b. However, if b comes first to the process, it is saved by the `save *` construct and is consumed in the next state.

Events in a coregion are ordered by the Synthesizer using the following rules in this order:

1 All active events are executed (that is, message output, timer set and reset, instance creation and action events).

2 All stimuli are executed (that is, message input and timeout events).

3 If a coregion contains a creation of process P and a message output to P, then any creation of P is executed before any message is output to P.

There are two methods for code generation for coregions. Generation of active event permutations may be enabled or disabled.

- If permutation generation is *enabled:*
  - a*ll* sequences of active events satisfying condition (3) above are generated
  - a*t least one* sequence of stimuli is generated
- If permutation generation is *disabled:*
  - a*t least one* sequence of active events is generated, and
  - a*t least one* sequence of stimuli is generated

By default, the permutation generation is disabled. To enable it, use the –C command line option or the following line in msc2sdl.ini:

```
OPT_EXPAND_COREGIONS = yes
```

Note: It is possible that only one sequence of stimuli in a coregion will be generated. Normally, this does not restrict functionality of the generated SDL process. This is achieved due to save * constructs in SDL states. Let's say, for instance, a coregion contains inputs of messages "a" and "b". The Synthesizer may generate their input in some fixed order. For example, it may first generate "a" and then "b". However if "b" comes first in the process, it is saved by the save * construct and is consumed in the next state.

CHAPTER 7

Synthesis: Compositions of multiple MSCs

In This Chapter

Specifying more complex behavior

To specify more complex behavior, for example behavior that involves alternatives or repeatable events, one should use compositions of multiple basic MSCs. The KLOCwork MSC to SDL Synthesizer supports three ways for describing composition of MSCs:

- implicit composition
- high-level MSCs (HMSC), or
- global conditions

HMSCs are preferred because they visualize the roadmap of several scenarios and allow reuse of an MSC several times in the specification.

The composition rules are different for HMSCs and global conditions.

Note: If an input model contains at least one HMSC, the HMSCs composition rules are applied. Otherwise, the rules for global conditions are applied.

The following sections describe each type of composition in more detail.

Implicit composition

When the input model consists of several basic MSCs, they are considered as *alternative behaviors* of a set of actors. For example, the two MSCs in the following figure (msc Success and msc Fail) define two possible scenarios for the set of two actors (instances Client and Server).

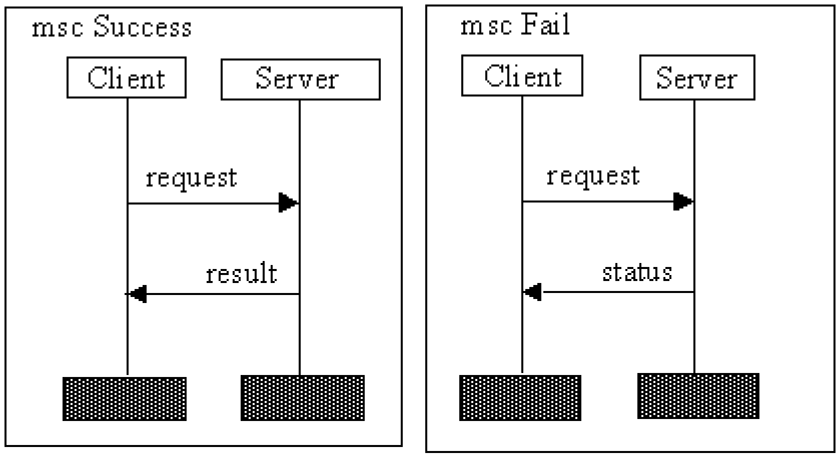


Figure 26: Implicit composition of two MSCs

Composition using HMSCs

High-level MSCs provide a means to graphically define how a set of MSCs should be combined. An HMSC is a directed graph in which each node is one of the following things:

- a start symbol (there is only one start symbol in each HMSC)
- an end symbol
- an MSC reference
- a condition
- a connection point

The flow lines connect the nodes in the HMSC and they indicate the sequencing that is possible among the nodes in the HMSC. The incoming lines are always connected to the top edge of the node symbol, whereas, the outgoing flow lines are connected to the bottom edge. If there is more than one outgoing flow line from a node this indicates an alternative.

The MSC references can be used to reference a single MSC. The conditions in HMSCs can be used to indicate global system states or guards. The connection points are introduced to simplify the layout of HMSCs and have no semantic meaning.

```
<hmsc diagram> ::=  <msc symbol>
            contains <msc heading> <mscexpr area>
<mscexpr area> ::= { <text layer>
                  | <start area>
                  | <node expression area>*
                  | <hmsc end area> * } set
<start area> ::= <hmsc start symbol>
            is followed by  { <alt op area>+ } set
<hmsc start symbol> ::=         ▽
<hmsc end area> ::= <hmsc end symbol>
            is attached to { <hmsc line symbol>+ } set
<hmsc end symbol> ::=           △
<hmsc line symbol> ::= <hmsc line symbol1>
                   | <hmsc line symbol2>
<hmsc line symbol1> ::=               │

<hmsc line symbol2> ::=               ↓
```

```
<alt op area>::= <hmsc line symbol>
     is attached to {<node area> | <hmsc end symbol>
}
<node expression area> ::=
     is followed by {<alt op area>+ } set
     is attached to {<hmsc line symbol>+ } set
<node area> ::= <hmsc reference area>
                       | <connection point symbol>
                       | <hmsc condition area>
<hmsc reference area> ::= <msc reference symbol>
                  contains <msc name>
<connection point symbol> ::=    ○
<hmsc condition area> ::= <condition symbol>
                        contains <condition text>
```

The KLOCwork MSC to SDL Synthesizer uses the following rules for HMSC analysis.

1 Each MSC reference in HMSC must point to a simple MSC (not to an HMSC). This MSC must be present in the input to KLOCwork MSC to SDL Synthesizer.

2 If a simple MSC is not mentioned in some HMSC, it is ignored.

3 All conditions with the same name on HMSCs are identified. For example, HMSC H3 in ***Sequential behavior with HMSC*** (see "Figure 27: Sequential behaviour with HMSC" on page 67) defines a loop.

4 If several HMSCs are specified in the input, they are merged. Their start symbols are joined together. Rule 3 above for conditions is then applied for the resulting HMSC.

5 If HMSCs are used, only local decisions in simple MSCs are analyzed. See ***Explicit local decisions based on data*** (on page 88). All other conditions are ignored.

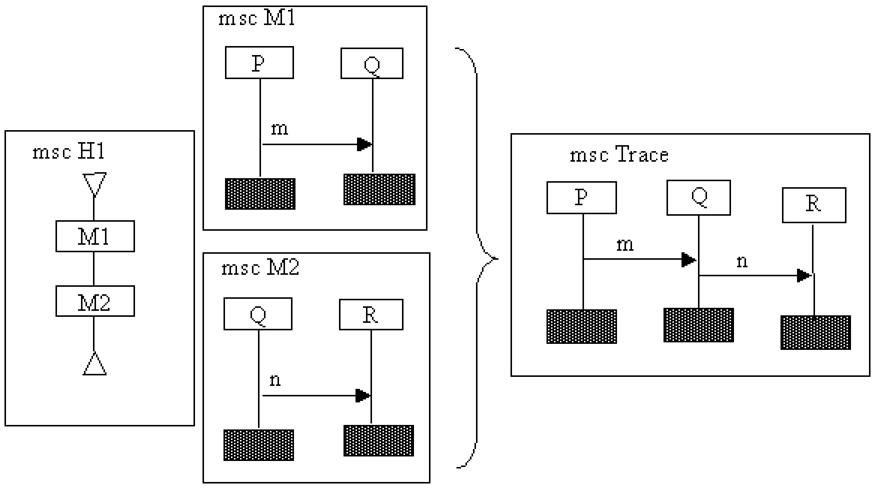
Let's consider several examples of composition using HMSC.

By using sequences of MSC references, one can define *sequential behavior*. Consider for example ***HMSC H1*** (see "Figure 27: Sequential behaviour with HMSC" on page 67). HMSC H1 defines a sequence of MSCs M1 and M2.

The input scenario model (HMSC H1 and MSCs M1 and M2) define a system with three actors (P,Q and R). The behavior is defined as follows:

- First, actor P sends message m to actor Q
- Second, actor Q sends message n to actor R

Figure 27: Sequential behaviour with HMSC

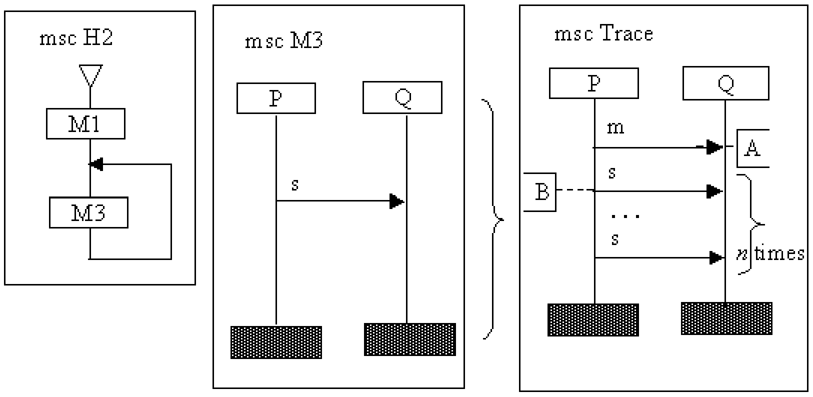


The composition of MSCs M1 and M2 is performed as follows. Consider a restricted case when no instance creation is used in MSCs.

Note: The axes with equal names in M1 and M2 are concatenated. If an axis name is present in only one of two MSCs, this axis is copied to the resulting MSC.

For example, the composition of MSCs M1 and M2 in ***Sequential behavior with HMSC*** (see "Figure 27: Sequential behaviour with HMSC" on page 67) is equivalent to MSC Trace. By using cycles in HMSCs, one can represent *repeatable behavior*. Consider, for example, HMSC H2 in ***Repeatable behavior with HMSC*** (see "Figure 28: Repeatable behavior with HMSC" on page 68). HMSC H2 defines repetition of the MSC M3.

Figure 28: Repeatable behavior with HMSC



Repeatable behavior with HMSC (see "Figure 28: Repeatable behavior with HMSC" on page 68) defines a system with two actors (P and Q) in which

- Actor P sends message m to actor Q, and then
- Actor P repeatedly sends message s to actor Q.

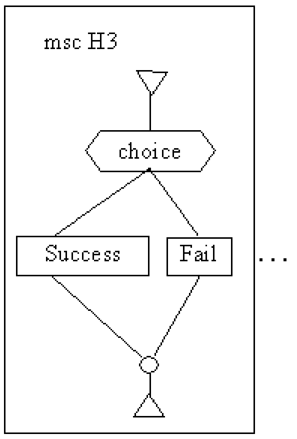
There is no single basic MSC that is equivalent to HMSC H2 (as opposed to the previous example). The history of the system's functioning after *n* transmissions of the message s is represented at MSC Trace. Note, also, that the ordering between events B (output of s) and A (input of m) is *not* specified by H2. It is possible that event B occurs *before* event A.

By using alternative flowlines in HMSCs, one can specify alternative behavior. Consider for example HMSC H3 in ***Alternative behavior with HMSC*** (see "Figure 29: Alternative behavior with HMSC" on page 69). HMSC H3 defines MSCs Success and Fail as two alternative behaviors. HMSC H3 contains a condition with the name choice that represents the decision point in the behavior. MSCs Success and Fail are described in ***Implicit composition of two MSCs*** (see "Figure 26: Implicit composition of two MSCs" on page 64).

Note: HMSC H3 is equivalent to implicit composition of MSCs Success and Fail.

There is no single MSC that is equivalent to HMSC H3.

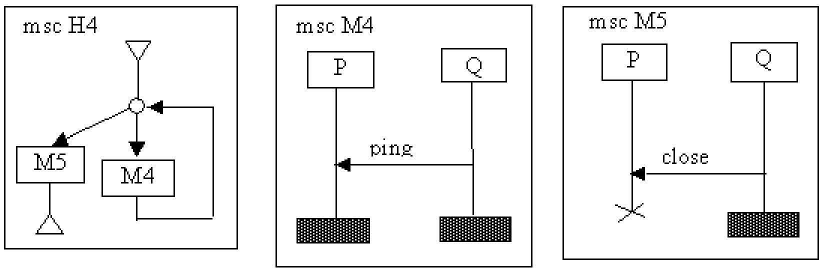
Figure 29: Alternative behavior with HMSC



By using a combination of the above techniques, one can represent *arbitrary behaviors*. Consider for example HMSC H4 in ***Loop with an alternative and an exit*** see "Figure 30: Loop with an alternative and an exit" on page 70. HMSC H4 defines a system consisting of two actors (P and Q). The behavior of the system is a repetition of sending a message ping to actor P by actor Q (MSC M4), optionally followed by sending message close by actor Q to actor P (MSC M5).

Note: The connection symbol in HMSC describes alternative choice between MSCs M4 and M5. The same connection symbol is also the endpoint to the cyclic flowline.

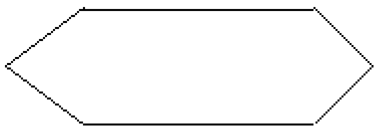
Figure 30: Loop with an alternative and an exit



Composition using conditions

Global conditions provide another way to specify control flows in MSCs (an alternative to HMSCs). The simplest and most important case of a global condition is a condition, which shares all instances on the MSC diagram. A condition may be considered as global in a few more cases (described in ***What conditions are considered global?*** on page 75) Initial conditions are further described in ***Initial conditions*** (on page 74).

```
<condition area> ::=  <condition symbol>
            contains <condition name> [ <shared> ]
            is attached to { <instance axis symbol> *}
<condition symbol> ::=
```



```
<condition name> ::= <name>
<shared> ::= shared { [ <shared instance list> ] | all }
<shared instance list> ::= <instance name>
                   [ , <shared instance list> ]
```

The <condition area> may refer to just one instance, or is attached to several instances. If a shared <condition> crosses an <instance axis symbol> which is not involved in this condition the <instance axis symbol> is drawn through.

Global conditions are used to define behavior as follows. Informally speaking, a process may "continue" from any global condition to another one with the same name. In other words, the following rule is used:

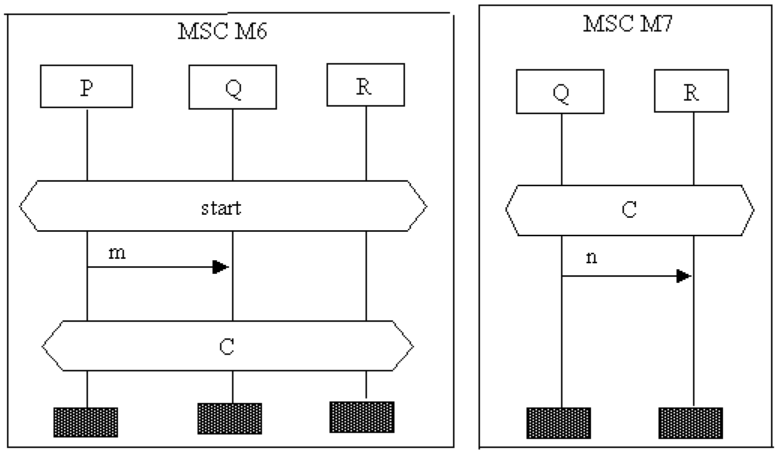
If on MSCs M1 and M2 instance P intersects a global condition C, the behavior of P before C on M1 may be continued by behavior of P after C on M2. (In general, M1 may coincide with M2).

The global condition with the name `START` denotes the beginning of the system behavior.

Let's consider some examples of specifying complex behavior using global conditions.

MSCs M6 and M7 in ***Sequential behavior with conditions*** (see "Figure 31: Sequential behavior with conditions" on page 72) are equivalent to HMSC H1 in ***Sequential behavior with HMSC*** (see "Figure 27: Sequential behaviour with HMSC" on page 67) and demonstrate how *sequential behavior* is specified with conditions.

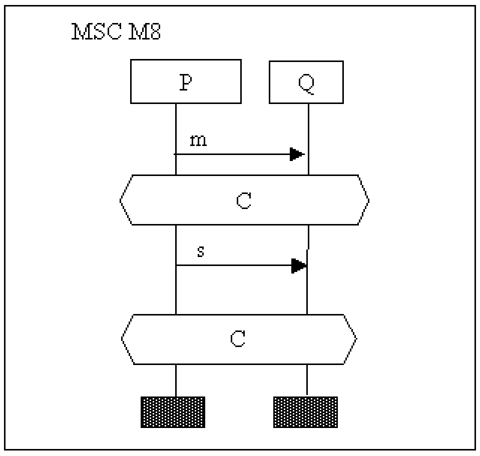
Figure 31: Sequential behavior with conditions



An empty line connects `START` and `C` for R in M6. If this line did not exist, there would be no way for R from `START` to `C` and the input of *n* in M7 would be lost.

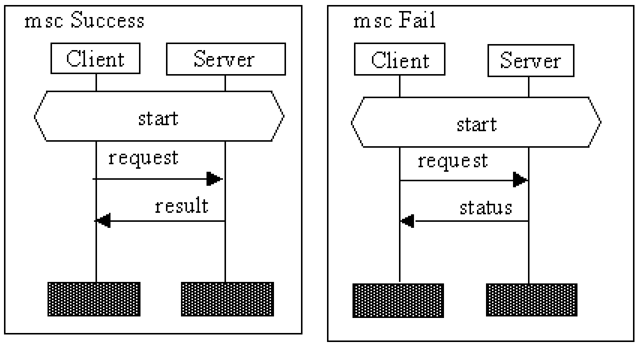
Specification of *repeatable behavior* in the following figure is equivalent to HMSC H2 in ***Repeatable behavior with HMSC*** (see "Figure 28: Repeatable behavior with HMSC" on page 68).

Figure 32: Repeatable behavior with conditions



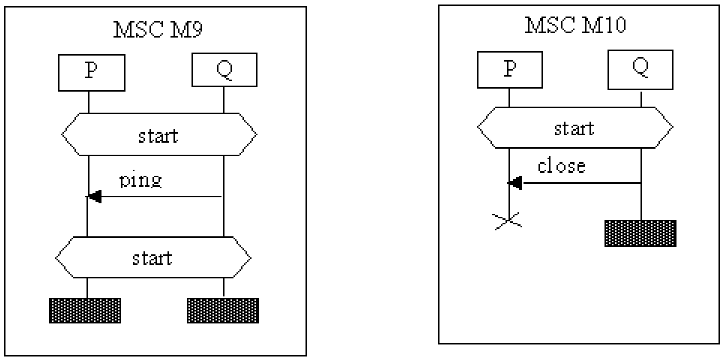
Specification of *alternative behavior* in the following figure is equivalent to HMSC H3 in ***Alternative behavior with HMSC*** (see "Figure 29: Alternative behavior with HMSC" on page 69).

Figure 33: Alternative behavior with conditions



The above rules can be applied to specify arbitrary behavior. For example, composition of MSCs M9 and M10 in ***Loop with an exit option with conditions*** (see "Figure 34: Loop with an exit option with conditions" on page 73) is equivalent to HMSC H4 in ***Loop with an alternative and an exit*** see "Figure 30: Loop with an alternative and an exit" on page 70.

Figure 34: Loop with an exit option with conditions



Initial conditions

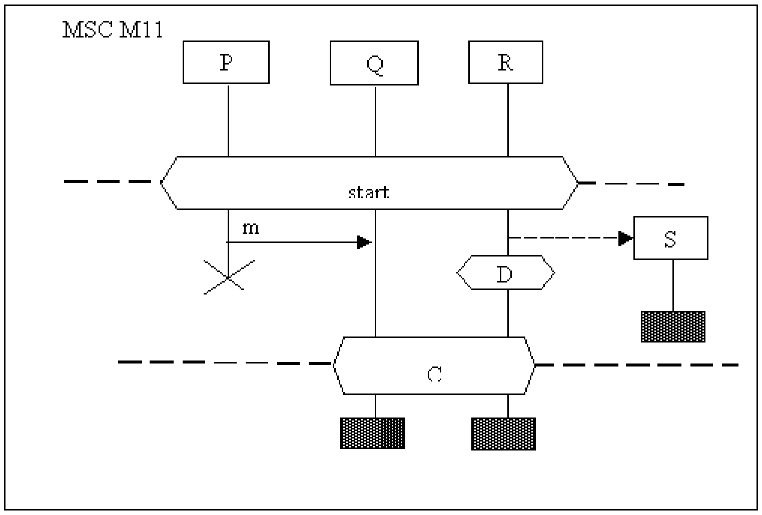
Condition is called *initial* if each axis on the MSC

- has this condition at the beginning, or
- this instance is created.

For example, condition `START` is initial on MSCs M6 in ***Sequential behavior with conditions*** (see "Figure 31: Sequential behavior with conditions" on page 72) and MSC M7 in ***Global and local conditions*** see "Figure 35: Global and local conditions" on page 74. (The bold dashed lines in Global and local conditions are required in the next subsection). The KLOCwork MSC to SDL Synthesizer uses the following rule:

If an MSC is not empty and has no initial condition, the initial condition `START` is added to it.

Figure 35: Global and local conditions



What conditions are considered global?

The KLOCwork MSC to SDL Synthesizer uses the following rule to find the global conditions:

> The MSC may be transformed so that the global conditions split it into parts by horizontal lines. These lines intersect no symbols except the corresponding conditions.

For example, conditions `start` and `c` in MSC M11 (***Global and local conditions*** see "Figure 35: Global and local conditions" on page 74) are global but `D` is not. The bold dashed lines in Global and local conditions show how the global conditions split the MSC. Therefore it is possible to compose M11 and M7 by means of condition C.

As a rule, it is impossible to add new conditions to the found set of global conditions so that the property in the frame is not violated. In particular, every condition that shares all instances on the diagram and is not a local decision is global.

CHAPTER 8

Synthesis: End of instance handling

The control flow end of the input MSC specification is defined as follows:

- when HMSC is used, it is the HMSC end symbol, or
- when global conditions are used, it is an endinstance symbol after which no events on another MSC may follow.

The control flow end may be handled in three different ways depending on synthesizer options. These options are specified as follows:

- Command line options:

  -e *type*, --end *type*
- Lines in msc2sdl.ini file:

  OPT_END_HANDLING = *type*

String *type* here specifies the way of handling the control flow end. It may take the following values:

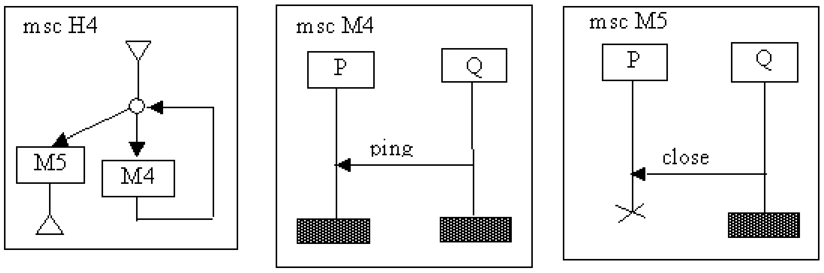
1. type = **state**: In this case when a process reaches the control flow end it passes to a state where it doesn't accept any signals and doesn't perform any actions. This is the default option.
2. type = **stop**: In this case when a process reaches the control flow end it stops (that is, executes a stop symbol).
3. type = **merge**: In this case alternative branches ending at the control flow end are merged. If one branch contains another one at the beginning the shorter branch is ignored. This may be useful when several simple MSCs are used for synthesis, which are execution traces of some system. Suppose that for some of them their recording have been interrupted before the system finished its work. Then endinstance symbols may mean the end of the system *observation* rather than the end of its *functioning*. If the behavior after an endinstance symbol is defined on some longer trace it would be reasonable to ignore this endinstance symbol. This may be achieved by using **merge** option for handling the control flow end.

CHAPTER 9

Synthesis: Non-determinism considerations

In many cases input MSCs do not specify deterministic behavior for all or some instances. Consider, for example, MSCs in the following figure:

Figure 36: Non-deterministic scenario



Here process Q may send either a 'ping' or a 'close' message. We don't know from the MSCs what the choice between them depends on.

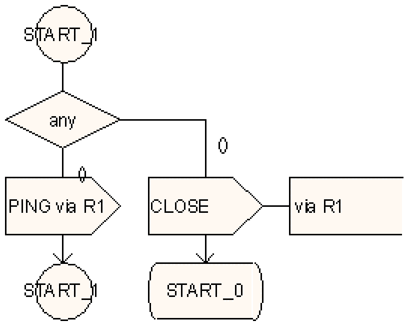
The KLOCwork MSC to SDL Synthesizer treats such branching as non-deterministic. It normally uses the SDL construct `decision any` to implement it. For example, for process Q the following code fragment would be generated:

Plain SDL

Phrase representation

```
connection START_1:
  decision any;
  ():
  output CLOSE via R1;
  nextstate START_0;
  ():
    output PING via R1;
      join START_1;
  enddecision;
```

Graphical representation



The following general rule for handling non-determinism applies. Let's call *stimuli* message inputs and timeout events. All events that are neither stimuli nor local decisions are called *active events*. They include actions, timer set and reset events, create and stop events. In general, `decision any` constructs may be generated in the following cases:

- When there are several alternative active events (as in the example in ***Loop with an alternative and an exit*** see "Figure 30: Loop with an alternative and an exit" on page 70).
- When there are alternative active events and stimuli.
- When there are alternative local decisions and other events. In this case a non-critical warning is reported for the `decision any` construct.

CHAPTER 10

Synthesis: Slices of MSCs

Slices are used to synthesize the SDL model from a subset of instances rather that the complete set of instances in the input MSC specification. This subset (the so-called *slice*) is defined as follows:

- The user selects some MSC instances
- The selected set of instances form the *slice* of the input MSC specification
- All other instances will be called *external* instances.

The generated SDL model will contain processes only for the instances from the slice. The events from the external instances are ignored. These external instances are handled as the system *environment*. Message exchange with them is interpreted as message exchange with the environment. That is,

- a message output to an external instance is interpreted as output to the environment,
- a message input from an external instance is treated similarly.

If an instance from the slice creates an external instance a message output is generated instead of process creation. The message name is `CREATEPROC_x` where "x" is the created instance name.

In order to include instances <inst1>,…, <instn> to the slice one should

- use -S <inst1> … -S <instn> options in the command line, or
- include the following line to msc2sdl.ini file:

SLICE = "<inst1>…<instn>"

By default no slice is specified. This is equivalent to a slice containing *all* instances.

CHAPTER 11

Synthesis: Data manipulations in MSCs

In This Chapter

Introduction

The KLOCwork MSC to SDL Synthesizer performs data manipulations within the framework of the MSC scenario language, for example:

- declarations of variables
- actions (see ***Action*** on page 53)
- messages with parameters (see ***Message*** on page 44)
- create events with parameters (see ***Instance creation*** on page 55)
- local decisions on data

This allows adding code in SDL to MSCs and describing operations on data.

Automatic data declarations

Every signal used in the synthesized SDL model must have a declaration.

The KLOCwork MSC to SDL Synthesizer generates signal definitions semi-automatically using the information given in the specification. Here the derivation algorithm for signal parameter sorts is described. The same algorithm is used for finding sorts of process formal parameters.

Suppose in the source MSCs signal S is used with parameters. The KLOCwork MSC to SDL Synthesizer finds parameter sorts in the following way.

Consider two cases.

1. S gets its description (its signature) in **signal** sentence following **data** keyword in a text symbol. This description is transmitted to the output file. For process formal parameters **process** sentences are used.
2. S has no explicit description of its parameters. Then its signature is found out by the context of MSC document.

To do this, the KLOCwork MSC to SDL Synthesizer derives the type of each parameter.

1. If assignment to a variable or a single variable is used as the parameter, and this variable is described in a text symbol after **data** sentence, the type of this variable is identified with the type of the signal parameter.
2. Otherwise, it is assumed that the parameter type is ENUM. ENUM is the unique type for all signal or process parameters matching (2b). All constants used as values of these parameters are considered as literals of ENUM. If the parameter type is defined according to (1) or (2a) then the names involved to the signal instances are not considered as literals of ENUM.

Examples:

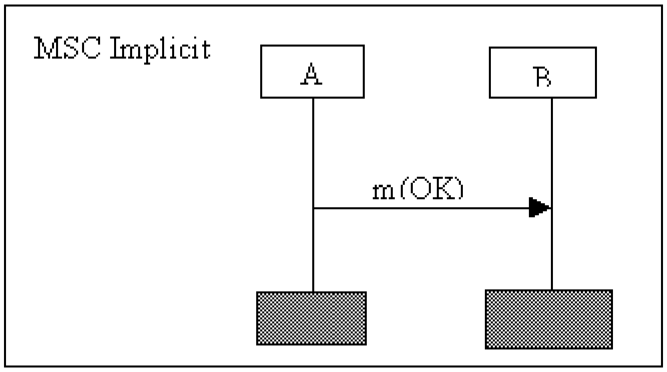


Figure 37: Automatic derivation of declarations for undefined parameter of message

In the preceding figure, rule (2b) is applied. The definition of signal m is the following:

```
signal m(ENUM);
newtype ENUM
   literals OK;
endnewtype;
```

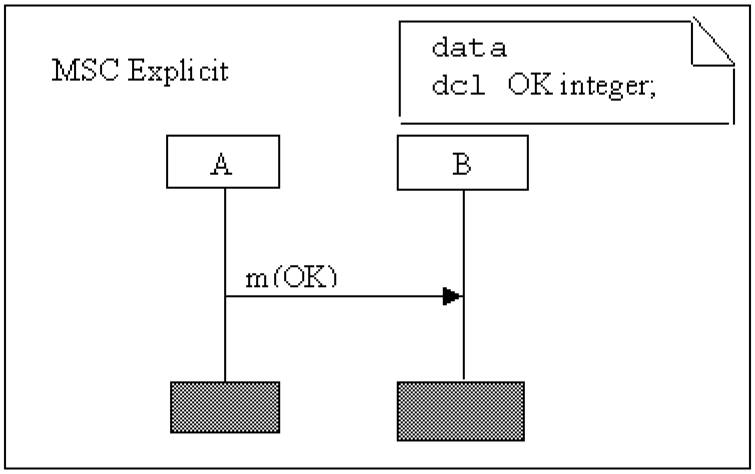


Figure 38: Automatic derivation of declarations for defined parameter of message

In the preceding figure, **signal** m has INTEGER as the type of its single parameter as defined in `dcl` clause:

```
signal m(INTEGER);
```

Explicit data declarations

Contents of a text symbol have the following syntax.

```
<text-symbol-contents> ::=
data <declaration>*
      | process <text>
      | actor <name> <text>
      | system <text>
      | use <import-list>;
      | block <text>
      | <text>
```

The <text> here is any text. The last alternative describes the case in which there is no keyword `data, process, actor, system, use` or `block` at the beginning of the text inside the symbol. In this case, the text symbol is ignored.

If the text inside the symbol begins from `data` keyword, it may contain declarations of variables, signals or processes according to the following syntax:

```
<declaration> ::=
      dcl <variable name> {, <variable name> }* <sort
name>;
   | signal <signature> {, <signature> };
   | process <signature> {, <signature> };
<signature> ::= <name> [ ( <sort name>, {<sort name>}* )
]
```

<sort name> is a name of any SDL sort.

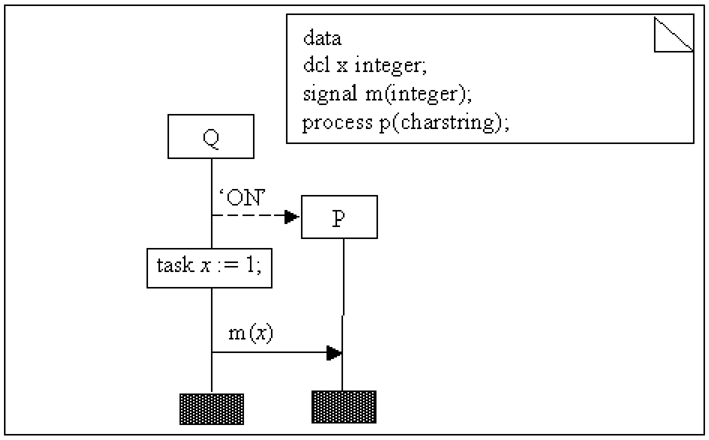
Every variable defined in a `dcl` construct is placed to all processes for MSC instances.

The `signal` or `process` signatures define types of signal of process parameters. In absence of signatures, the Synthesizer derives parameter types implicitly.

After `use` keyword in text symbols follow external package names. These packages are imported into the generated system (in plain SDL) or the package (type-based SDL): `<import-list> ::= <package name> {, <package name> }*`

Figure 39: Process and signal parameters are defined explicitly

Example:



The `process, block, system,` and `actor` keywords in `text symbols` are used to specify SDL code that is directly copied to the output file. The code is denoted as <text> in BNFs and is not analyzed by MSC to SDL Synthesizer.

Depending on the keyword, the code is placed in different places within the synthesized system.

`process <text>`	– <text> is generated into processes for all instances
`actor <name> <text>`	– <text> is generated into process for instance <name>
`block <text>`	– <text> is generated on the block diagram showing interaction between instances
`system <text>`	– <text> is generated into the system

Note: The KLOCwork MSC to SDL Synthesizer does not take information about variables, signals and processes from SDL code specified in this way. Therefore, it is often preferable to use the `data` keyword to describe variables, signals, and process parameters.

Explicit local decisions based on data

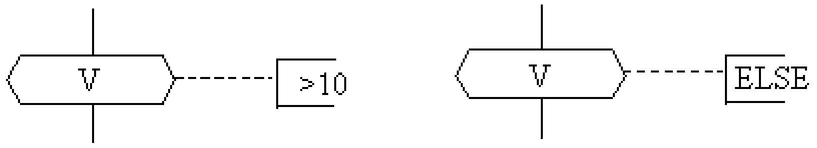
MSC to SDL Synthesizer supports specification of local decisions, which check values of process variables. Syntactically, local decisions are specified as local conditions in the corresponding instance. Each decision contains a variable *name* and a *guard*. Variable on the values of which decision is made is written in the condition symbol. The guard specifies a condition on the variable value. It is written in a comment attached to the local condition.

The syntax of the guard is as follows:

```
<guard> ::= <answer> | else
```

in which <answer> is any <answer> clause of SDL (such clauses describe alternative decisions, (see [Z.100, 2.7.5]). Examples of local decisions are:

Figure 40: Example of explicit local decisions

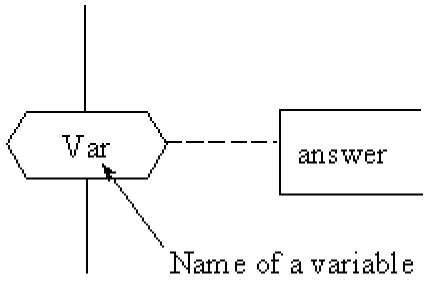


The semantics of such a condition are as follows. During runtime, a process may pass through it only when the value of the Boolean expression is true. Alternative sequences of events can be specified in different MSCs using local conditions with the same name and different guards.

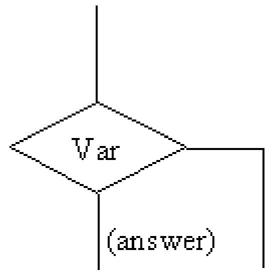
Guard `ELSE` defines the default event sequence, which is executed when other alternatives evaluate to false.

In general, the following decision

Figure 41: Explicit local decision



is mapped to the following SDL:

Phrase representation	Graphical representation
... decision Var: ... (answer): ...	

The syntax of local decisions in MSC PR files is as follows (apostrophes should be doubled inside <guard> after `comment` keyword when textual syntax is used).

```
<local decision> ::=
      condition <variable name> comment '<guard>';
```

The following rules for writing decisions are recommended:

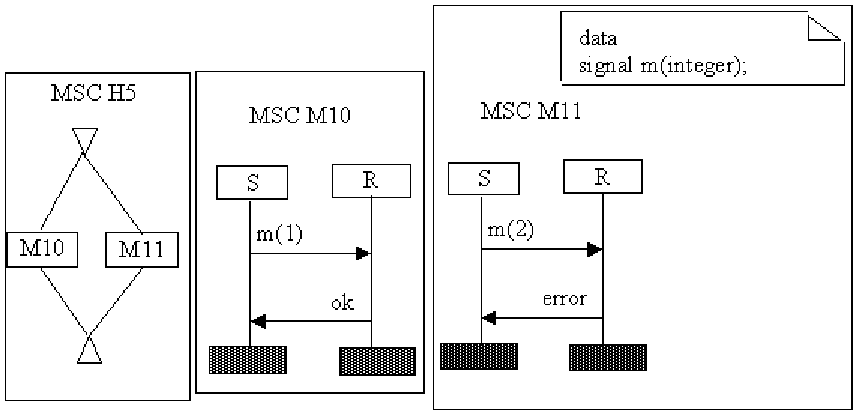
1 Alternative local decisions must have the same variable name. Their guards should be mutually exclusive.

2 Suppose that HMSCs are used. Then local decisions in different MSCs, which describe alternative branches, must appear either immediately after instance heads or after equivalent event sequences after instance heads. References to these MSCs must follow the same symbol on the HMSC (for example, the same condition).

Implicit local decisions based on data

Consider now the case when some parameters of a message are constants. In the following diagram, instance S chooses 1 or 2 as the parameter of message m. Instance R receives m and depending on the parameter value chooses its further actions. If the received value is 1 R sends the message 'ok', otherwise it sends 'error'.

Figure 42: Parameters of a message are constants



The syntax of constants is as follows:

```
 <constant> ::=
        <literal name>
      | <integer constant>
      | <real constant>
      | <character string constant>
      | <boolean constant>
<boolean constant> ::= true | false
<integer constant> ::= [ + | - ] <digit> +
<real constant> ::= [ + | - ] ( <digit> * . <digit> + |
<digit> + . <digit> * ) [ ( E | D ) <integer constant> ]
<character string constant> ::= ' <text> ' | " <text> "
```

Consider the case in which a message has one parameter. The behavior of the sending and receiving processes is defined as follows. Let M be the message name. If the message parameter is a constant, the sender simply sends it. The receiver, after receiving the message M, proceeds as follows. Consider all alternative behavior branches for the receiver after reception of M. For example, in the preceding figure, process R replies either with 'ok' or with 'error'.

Consider now the following three cases.

1 The received value matches a constant parameter in exactly one branch. Then the receiver executes this branch. For example, for MSC H5 process R replies with 'ok' to 'm(1)' and with 'error' to 'm(2)'.

2 The received value does not match any constant (the condition of the previous point is false), but there is one branch with a variable or an assignment in the parameter. That branch is executed. For example, for MSC H6 in the following figure, process R replies with 'unknown' if it receives m(x) with x not equal to 1.

3 In all other cases, the KLOCwork MSC to SDL Synthesizer defines the behavior of the receiver by its built-in algorithm.

For messages with more than one parameter, the constants are handled by generalization of the algorithm described above.

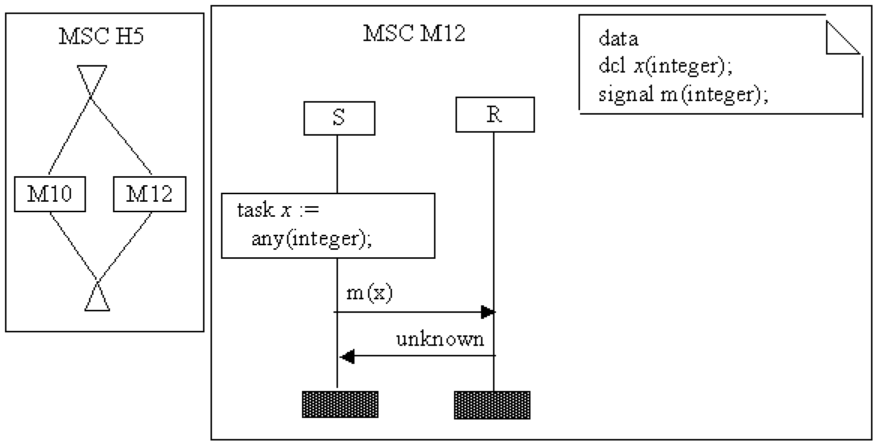


Figure 43: HMSC and MSC diagrams of messages with parameters

CHAPTER 12

Example of an MSC specification: A simple network

In This Chapter

Description of the simple network

This section describes an example of an MSC specification. It also describes the steps for jump-starting an SDL project within Telelogic Tau using the KLOCwork MSC to SDL Synthesizer.

The MSC specification describes a simple network that provides connections between subscribers. The network provides services for initiating and terminating connections. Connection can be terminated only by the same subscriber that initiated it. During connection initiation, an abnormal situation may occur when the called subscriber does not reply within a certain time.

MSC specification

The network is represented on MSC diagrams by the instance *Network*. The instance *Client* represents the active subscriber who initiates and terminates connections. The instance *Server* represents the passive subscriber to which the Client connects.

Each MSC specification describes a single communication session between *Client* and *Server*. The session includes connection initiation and termination. It progresses according to the following scenario:

- A connection request is sent from *Client* to *Network* and is passed to *Server* as shown on MSC Connect_Request.
- *Network* gets a response from *Server* before the timeout expires. In this case a connection is initiated successfully. This is described in MSC Connect_Establish.
- The following abnormal situation can also occur: *Client* doesn't reply during the specified time interval as described on MSC Connect_Timeout. In this case, connection fails.

All the MSC diagrams shown in this section are used as input to the KLOCwork MSC to SDL Synthesizer.

One of them, HMSC diagram for Network example, is an HMSC. All the others are simple MSCs. The HMSC defines the sequence of MSC composition (or, in other words, sequence of their execution). In the rest of this subsection these MSC diagrams are described. Note that, as an HMSC is present in the specification, the global condition in simple MSCs have no semantics and play the role of comments.

The first ***MSC diagram Connect_Request*** see "Figure 45: MSC diagram Connect_Request" on page 95 describes a request for a connection from *Client* to *Server* through the Network. *Client* sends message *ConnectReq* to Network and Network passes it to *Server*. MSC starts with condition `start` and finishes with condition `wait`, after which Network waits for *Server* to respond.

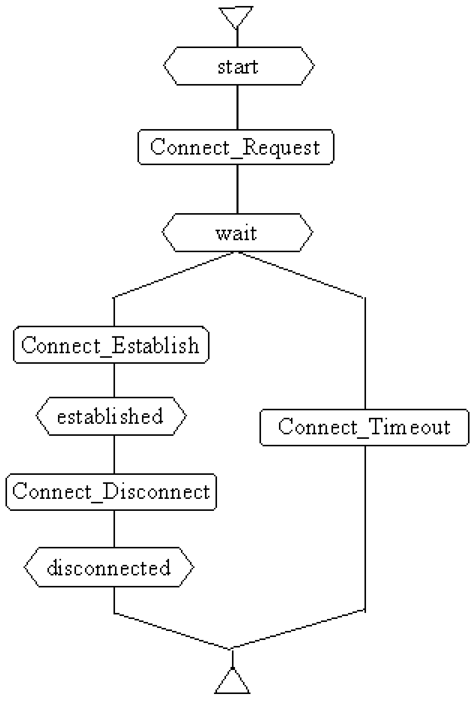


Figure 44: HMSC diagram for Network example

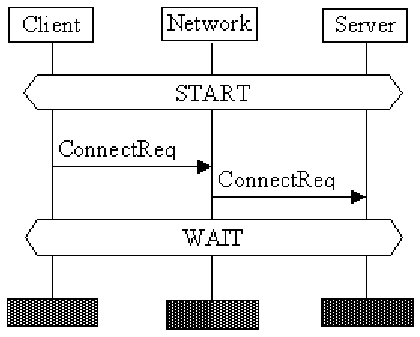
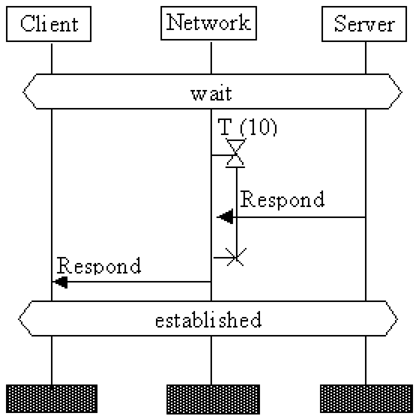


Figure 45: MSC diagram Connect_Request

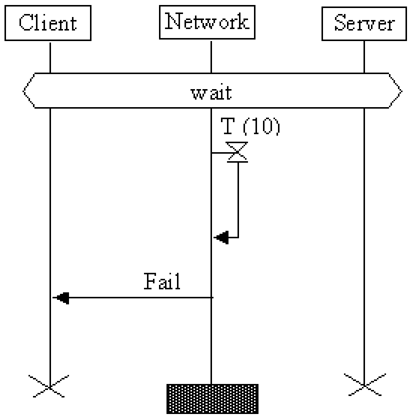
The next MSC is Connect_Establish. On this MSC *Network* receives *Respond* signal from *Server* before the timeout expires and passes the *Respond* signal to the *Client*. After this the connection is considered established and all actors go to the global condition `established`.

Figure 46: MSC diagram Connect_Establish



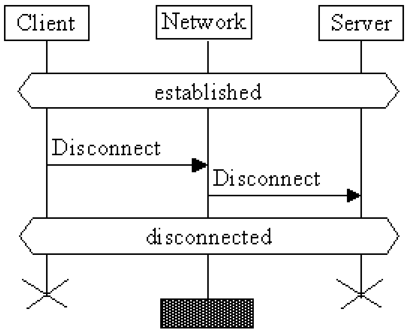
MSC Connect_Timeout describes abnormal behavior of the model. If *Client* does not reply within the specified time, *Network* sends the message *Fail* to the *Client* and connection fails. After this, *Client* and *Server* instances are stopped, but the behavior of the *Network* is unspecified.

Figure 47: MSC diagram Connect_Timeout



To end the connection, *Client* sends the message *Disconnect* to *Network*, and *Network* passes this message to *Server*. After this connection is ended (condition `disconnected`) and *Client* and *Server* instances stop,. *Network* instance does not stop, but its behavior is unspecified in this case too. It is described in ***MSC diagram Connect_Disconnect*** (see "Figure 48: MSC diagram Connect_Disconnect" on page 97).

Figure 48: MSC diagram Connect_Disconnect



The model and diagram files for Telelogic Tau describing the Network example may be found in the doc/examples/Network subdirectory of the MSC to SDL Synthesizer installation directory.

CHAPTER 13

MSC PR syntax

In This Chapter

Input language

The input language, supported by MSC to SDL Synthesizer is a combination of the phrase representation syntax of Message Sequence Charts (MSC PR) published in Z.120 standard by ITU in 1992 and 1996. Some supported extensions to this language are described in the latest edition of the standard, the so-called MSC-2000. Some constructs of the MSC language are not supported (refer to ***Known limitations*** on page 115). Complete syntax rules of the language accepted by MSC to SDL Synthesizer are presented below in BNF notation.

Keywords

```
<keyword> ::=
      action
      | all
      | alt
      | begin
      | block
      | comment
      | concurrent
      | condition
      | connect
      | create
      | decomposed
      | end
      | endconcurrent
      | endinstance
      | endmsc
      | endmscdocument
      | env
      | exc
      | expr
      | found
      | from
      | in
      | inst
      | instance
      | instancehead
      | loop
      | lost
      | msc
      | mscdocument
      | name
      | opt
      | out
      | par
      | process
      | reference
      | related
      | reset
      | seq
      | service
      | set
      | shared
      | stop
      | system
      | text
      | timeout
      | to
```

Lexical rules

```
 <apostrophe> ::= '
<quoted text> ::=
    <apostrophe>
     { <any character except apostrophe>
      | <apostrophe> <apostrophe> } *
      <apostrophe>
    | <any character except semicolon> *
<name> ::= { <letter> | <digit> | <underline> } +
<underline> ::= _
<digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
<letter> ::= a | b | ... | z | A | B | ... | Z
```

Syntax rules

```
<msc specification> ::=
      { <msc document>
      | <msc>
      | <hmsc> } *
<msc document> ::=
    mscdocument <msc document name>
                  [ related to <sdl reference> ] <end>
    { <msc> | <hmsc> } *
    [ endmscdocument <end> ]
<end> ::= [ comment <quoted text> ] ;
```

Note: An optional `related to` clause is supported for compatibility with MSC standard. (MSC'92 syntax is used.) The synthesis algorithm ignores it.

```
<sdl reference> ::=
    { <path item> / } * <name>
<path item> ::=
    { system | block | process } <name>
<msc> ::=
    { msc | submsc } <msc name> <end>
    [ <inst decls> ]
    { <condition> <end> } *
    { <msc statement> | <text definition> } *
    { <condition> <end> } *
    endmsc } <end>
```

Note: Initial and Final conditions around a sequence of <msc statement>s and <text definition>s are supported for compatibility with the MSC'92 standard.

```
<inst decls>  ::=
        inst <instance decl> { , <instance decl> }* <end>
<instance decl> ::=
        <instance name> [ : <instance kind> ]
<instance kind> ::=
    [  system |  block  | process | service  ]  <name>
```

Note: `Optional <instance kind>` is supported for compatibility with the MSC standard. (MSC'92 syntax is used.) The synthesis algorithm ignores it.

```
<event list> ::=
    { <event> <end> } +
<msc statement> ::=
      <sharing info> :  <event list>
    | <old instance>
<old instance> ::=
    instance <instance name> [ : <instance kind> ]
       [ decomposed ] <end>
```

Note 1: <old instance> is supported for compatibility with the MSC-92 standard.

Note 2: `decomposed` keyword is ignored by the synthesis algorithm.

```
<event> ::=
  <instance head>
| <message input>
| <message output>
| <create>
| <timer statement>
| <action>
| <stop>
| <endinstance>
| <concurrent>
| <endconcurrent>
| <condition>
| <not supported event>

<instance head>  ::=
    { instance  | instancehead }
    [ <instance kind> ] [ decomposed ]
```

Note: `instancehead` keyword is supported for compatibility with the MPR format of Telelogic Tau.

```
<message input> ::=
    in <msg ident> from <address>
<message output> ::=
    out <msg ident> to <address>
<msg ident> ::=
     <message name> [ , <message instance name> ]
                    [ <parameters> ]
```

Note: Syntax and semantics of <parameters> are described in ***Message*** on page 44 .

```
<address> ::=
       <name>
| env
| lost [ <name> | <env> }
| found [ <name> | <env> }
<condition> ::=
      condition <condition name>
```

```
        [ shared <sharing info> ]
    | <local decision>
<sharing info> ::=
    <instance name> { , <instance name> } *
    | all
```

Note: Interpretation of global conditions is explained in ***Composition using conditions*** (on page 71)

Note: `<local decision>` is an extension to the MSC language supported by MSC to SDL Synthesizer. Local decisions are described in ***Explicit local decisions based on data*** (on page 88) .

```
<create> ::=
    create <instance name> [ <parameters> ]
<action> ::=
    action <text>
```

Note: Analysis of `<text>` is described in ***Action*** on page 53 .

```
<stop> ::=
     stop
<endinstance> ::=
    endinstance
<concurrent> ::=
    concurrent
<endconcurrent> ::=
    endconcurrent
<timer statement> ::=
    <set>
    | <reset>
    | <timeout>
<set> ::=
    set <timer name> [ , <timer instance name> ]
                     [ <duration> ]
<duration> ::= ( <expression text> )
```

Note: <expression text> is any text with balanced brackets. Single and double quotes (" , ') are not regarded as string delimiters.

```
<reset> ::=
    reset  <timer name> [ , <timer instance name> ]
<timeout> ::=
    timeout <timer name> [ , <timer instance name> ]
<text definition> ::=
    text <Quoted text> <end>
```

Note: <text> may contain constructs (extensions) described in ***Explicit data declarations*** (on page 86)

```
<hmsc> ::=
      expr { <seq list> | '(' <seq list> ')' }
      { <label name> : <node> <end>
      | <text definition> } *
<node> ::=
      ( <msc name>
      | '(' <msc name> ')'
      | condition <condition name>
      | connect )
        seq '(' <seq list> ')'
      | end
<seq list> ::= <label name> { alt <label name> } *

<not supported event> ::=
      { reference <name>
      | { alt | exp | par | opt |
           loop [ < <name> [ , <name> ] > ] }
         [ begin [ <name> ] | end ]
      }
      [ shared <sharing info> ]
```

Note: Textual constructs in MSC PR matching <not supported event> are not analyzed and are ignored. However, syntax errors are not reported for them.

CHAPTER 14

Error messages

In This Chapter

The error messages, warnings and information messages from the KLOCwork MSC to SDL Synthesizer are listed below.

Warning messages are designated as critical and non-critical. Non-critical error messages concern less serious situations or are addressed to more experienced users.

By default, the KLOCwork MSC to SDL Synthesizer reports only critical warnings. To report non-critical warnings, do the following:

- If you are running Synthesizer integrated with Telelogic Tau, set the following option in the `msc2sdl.ini` file.

  ```
  OPT_ALL_WARNINGS=yes
  ```

- If you are running the Synthesizer from the command line, use the option `-w`.

References to source MSCs are output in the following way:

- When you run the Synthesizer from Telelogic Tau, references to source MSCs point (by default) to the symbols in GR diagrams. To use references to PR files, set the following options in the `msc2sdl.ini` file:

  ```
  OPT_CLEAN_GEN_PR=no
  ```

  ```
  OPT_CROSS_REFS=no
  ```

- When you run the Synthesizer from the command line, references to source MSCs point (by default) to the lines in PR files.

Error message list

E	ERROR message
W CR	WARNING Critical
W NC	WARNING Non-critical
W/I	WARNING or INFORMATION*

***Note:** Some messages are displayed as WARNING messages when the Synthesizer runs from Telelogic Tau and as INFORMATION messages when the Synthesizer runs from the command line. In the list below, these messages are labeled as W/I or WARNING/INFORMATION.

Error messages

E	All inline expressions are ignored (not supported)	Inline expressions are not supported by the KLOCwork MSC to SDL Synthesizer. They all are ignored.
E	All MSC references are ignored (not supported)	MSC references are not supported by the Synthesizer. They all are ignored.
E	Ambiguous reference to <name>.	The message or timer identification is ambiguous. An additional <message instance name> (or <timer instance name>) should be used to clearly identify message (or timer) inputs and outputs.
E	Cannot create temporary directory	The Synthesizer cannot create a directory where it stores MPR and SDL PR files. It is created in the target directory.
E	Cannot create output directory.	The Synthesizer cannot create a directory where it stores SDL GR files. It is created in the target directory.
E	Cannot write output file <name>.	The Synthesizer cannot write to file <name>.
E	Constant is used as data target.	A constant was used at the receiver side. A variable should be used instead of constant.

E	Could not get The MSC to SDL Synthesizer installation directory.	The Synthesizer cannot determine its installation directory. It may not have been installed properly.
E	Creation of environment instance is not allowed.	An creation event for instance with name env_0 occurs in some MSC. The creation of this instance is invalid as it represents the environment.
E	Duplicate CONCURRENT	A **concurrent** clause was met in a coregion.
E	Duplicate instance end	An **endinstance** or **stop** occurred after instance was finished.
E	Duplicate instance head	The instance contains two instance heads.
E	Duplicate label <name>	A <label name> in an HMSC is duplicate (see [GRAMMAR, HMSC])
E	Duplicate shared instance name <name>	A list of shared instance names contains a duplicate element.
E	Empty coregion.	A coregion must contain at least one event but it is empty.
E	Empty field in message parameters	The text of a message parameter is empty.
E	Error connecting to PostMaster	The Synthesizer failed to communicate with Telelogic Tau PostMaster program.
E	Error reading from PostMaster	The Synthesizer failed to communicate with Telelogic Tau PostMaster program.
E	Error reading <filename>: Diagrams other than MSC or HMSC are ignored.	The file extension of input documents for the Synthesizer must be msc (for simple MSCs) or mrm (for HMSCs). File names and extensions of documents are displayed in the Telelogic Tau Organizer window.
E	Error reading input files. You must select some MSCs.	User did not select any MSC in the Organizer.
E	Error reading input files.	On a PC, check that the directory name does not contain spaces.
E	Error: Unknown Code gen.	Code generator name specified after –t in the command line is invalid. The available names are: plainsdl and tbsdl.
E	Event after instance end	An event after **endinstance** or **stop**.
E	Found message is considered as received from ENV	The Synthesizer interprets lost messages as sent to the environment. This may differ from the meaning assumed by the user.

E	Instance <name> cannot be created twice	Several <create> events are used to create the same instance A.
E	Internal error	An internal error in the KLOCwork MSC to SDL Synthesizer. Please contact ***support@KLOCwork.com*** (see "mailto:support@KLOCwork.com - mailto:support@KLOCwork.com")
E	Invalid event ordering	The ordering relation between events is invalid. There may be a cyclic dependency between them.
E	Invalid event in a coregion	This event cannot occur in a coregion.
E	Invalid model name: <name>.	The specified SDL model name is invalid. It must be a string consisting of letters, digits and underscore characters.
E	Invalid multi instance event	Event shares several instances but event type does not allow this.
E	Lost message is considered as sent to ENV	The Synthesizer interprets found messages as received from the environment. This may differ from their meaning assumed by the user.
E	Missing CONCURRENT	**Endconcurrent** clause was met but **concurrent** clause is missing.
E	Missing ENDCONCURRENT in instance <name>	A coregion is not ended by **endconcurrent**.
E	Missing end of instance <name>	The instance is not ended by **endinstance** or **stop**.
E	Missing instance head	An event related to an instance is encountered but the instance head is missing.
E	MSC <name> redefined	The MSC with the name <name> is already defined.
E	No actors found	Input MSC files do not contain information about at least one actor.
E	No complementary statement for <name>	For a message of timer arrow only one end is specified.
E	No MSC-PR files specified.	At least one MSC PR file must be specified in the command line.
E	No start condition	There is no **start** global condition in the set of input MSCs.
E	Node is not reachable from the START symbol	There is no path from the START symbol in an HMSC to this node.

E	Number of signal parameters is different here WARNING/ INFORMATION and in this usage/definition.	Two occurrences of a signal/process have different number of parameters (Refer to (3), parameters of messages, Create command).
E	Prefix CREATE_ is not allowed in names of signals (it is reserved).	In the current implementation, the prefix CREATE_ is not allowed in signal names.
E	Prefix VAR_ is not allowed in names of variables (it is reserved).	In the current implementation, the prefix VAR_ is not allowed in variable names.
E	Redeclaration of <name> WARNING/ INFORMATION Previous declaration location.	A variable, signal or process <name> is redefined in a text symbol.
E	Reference to undefined label <label name>.	A <label name> in an HMSC is undefined.
E	Reference to undefined MSC <name>.	MSC <name> is not defined in the set of input MSCs.
E	Syntax error. WARNING/ INFORMATION expected tokens.	There is a syntax error in an input MPR file. The tokens expected by the analyzer are output in the WARNING (or INFORMATION) message.
E	Syntax error in data declaration.	A syntax error after DATA keyword in a text symbol.
E	Syntax error in message parameters.	The syntax of message parameters is incorrect.
E	<Telelogic Tau error text> Many PostMasters running. Could not decide which one to connect to.	Several PostMaster programs are running. PostMaster program is a part of Telelogic Tau. To avoid this error, terminate extra active PostMaster programs.
E	The model name is empty.	The specified SDL model name is an empty string.
E	Too many errors.	The maximal number of error messages was exceeded.
E	Too many warnings.	The maximal number of warning or information messages was exceeded.
E	Two definitions of instance <name> are not identical. WARNING/ INFORMATION Another definition of <name>.	The attributes of the instance do not match its previous definition.
E	Type of parameter <N> is different here WARNING/ INFORMATION of this usage/definition.	Parameter number <N> has different types in two occurrences of a signal or process.

E	Undefined instance <name>.	An instance with the name A is not defined.

Warning messages: Critical and non-critical

W NC	Condition C ignored.	Condition C is neither global nor local decision – such conditions are ignored by analyzer.
W CR	Condition C is not reachable by instance A.	It is impossible to reach global condition C from the START condition moving along the axis of instance A.
W NC	Condition intersects the following events: WARNING/ INFORMATION <event>.	Condition separates message output and input of timer set and reset/timeout.
W NC	Condition start is implicitly inserted at the beginning of the MSC <name>.	The MSC diagram has no initial global condition, therefore, a start condition is implicitly added to the beginning of the MSC.
W NC	Condition <name> ignored.	Condition with name <name> is ignored.
W NC	Condition <name> not reachable by instance <inst>.	Condition with name <name> is not reachable by instance with name <inst>.
W NC	Creation of an external instance was replaced by a message output	This warning appears when some instance in the slice tries to create new instance outside of the slice.
W NC	Data destination here conflicts with another event. WARNING/ INFORMATION The another event mentioned above.	The receiver can not decide in which of several different variables to store data. The Synthesizer leaves only one alternative and discards all others. For example, the following specification lead to this warning: alt signal m(x), signal m(y).
W NC	Event in environment is ignored	Reports that some events (actions, timers, stops) out of the slice are ignored.
W NC	Identifier name <word> is reserved. Renamed to <new-word>.	<word> is a keyword in SDL. The Synthesizer substituted <new_word> for the keyword.
W NC	Instance creation from environment is ignored	Appears when some instance outside of the slice tries to create new instance in the slice `(see Slices` "Synthesis: Slices of MSCs" on page 81`)`.

W NC	Local decision has a non-decision event as alternative WARNING/ INFORMATION Non-decision alternative to the local decision	There are alternative local decision and some non-decision event. An additional decision any SDL construct is generated here.
W NC	Local decision has a single alternative.	There is no alternative local decision for this decision. The events after it are, thus, always selected.
W CR	MSC <name> is not used.	There is no reference to MSC <name> from the HMSC.
W/I NC	No code generator specified.	No code generator (plainsdl or tbsdl) was specified when msc2sdl was called from the command line. The source MSC specification was checked for errors but no SDL was synthesized.
W NC	No continuation of condition <name> for instance <inst>.	The behavior of instance <inst> after global condition <name> is undefined.
W NC	Node is not reachable from the START symbol.	There is no path from the start symbol of an HMSC to this symbol.
W/I NC	Several input HMSCs were merged.	There are several HMSCs in the input. This message informs the user that according the the synthesis algorithm, the Synthesizer merges them.
W NC	Single alternative ignored.	There is no alternative local decision for this decision so the decision is ignored.
W NC	Variable was changed. WARNING/ INFORMATION because of conflict with this decision.	A variable name in local decision was changed because this decision conflicted with another one. They have different variable names.
W NC	Undefined variable: <name>.	Variable <name> is used in a local decision but is not defined after DATA keyword in a text symbol.

CHAPTER 15

Known limitations

In This Chapter

Unsupported MSC constructs

- In basic MSCs, MSC references and inline expressions are not supported.
- MSC references in HMSCs must point to simple MSCs (not to HMSCs).
- Parallel frames are not supported in HMSCs
- MSC references may contain only single MSC names.
- Parameters to MSCs are not supported
- Time constraints are not supported
- (H)MSC guards are not supported

Coregions

Set and reset in coregion can occur in reverse order

Telelogic Tau integration module

On UNIX, if SDL is synthesized successfully but MSC to SDL Synthesizer produces warning messages these messages remain for a very short time in the Organizer Log. They are removed by generating SDL GR files.

CHAPTER 16

Troubleshooting

Some possible problems arising in use of MSC to SDL Synthesizer are described below. Solutions for each possible problem are provided.

Problem	Solution
The synthesized SDL model doesn't match the input MSCs	This may occur because the intuitive meaning of MSCs assumed by the user differs from their interpretation by MSC to SDL Synthesizer. This interpretation is described in the section . Refer to this section and check the input MSC specification. From the MSC to SDL Synthesizer menu. select "`Analyze on`ly". The MSC specification is analyzed with non-critical warnings enabled. They may point to some semantic errors and warnings in the MSC specification.
Error messages in the Organizer Log refer to some SDL PR or MSC PR files. It is impossible to view the errors because these files do not exist.	Disable removal of generated PR files and repeat the synthesis.Use the option OPT_CLEAN_GEN_PR=no in `msc2sdl.i`ni file. See also ***Configuration file*** on page 30 .
Error messages in the Organizer Log advise you to correct MSC or SDL symbols in GR diagrams.	The real errors may be located in other symbols and both MSC to SDL Synthesizer and Telelogic Tau may be unable to correctly locate them. Try the following: enable references to PR files in error and warning messages (see ***Configuration file*** on page 30). Repeat the synthesis. The error messages should then contain references to PR files. Explore these PR files to find the errors.

Glossary of Terms

Actor
A coherent set of roles that users of the system play when interacting with the system

Basic MSC
An MSC that is not an HMSC and does not contain conditions and coregions.

Code generation
A process that produces a textual representation in selected target language from an internal representation

Coregion
A group of events of an MSC instance, in which no ordering is assumed

Diagram
A graphical presentation of a specification, rendered as a connected graph of vertices (symbols) and arcs (relationships)

Element
An atomic constituent of a model

Environment
Collection of all external actors for a particular system

Event
A specification of a significant occurrence that involves a certain MSC instance and has a location in time.

Global condition
A condition that is shared by all instances and is used for describing composition of scenarios or the control flow (Refer to ***Composition using conditions*** (on page 71)).

Graphical representation
Notation, describing the rendering of a specification as a two-dimensional collection of symbols and lines (a diagram)

HMSC
High-level MSC. Refer to ITU standards [Z120-96 (5.5), Z120-2000 (1.7.5)].

Initial condition
Condition that occurs before any other event on a given MSC for all instances

Instance
A part of an MSC model that is participates in MSC events and implements the behavior that conforms to the MSC specification

ITU
International Telecommunications Union – an international standards institution

Local condition
A condition that is shared by only one instance.

Local decision
A condition that is shared by only one instance and has a special comment attached to it. Special semantics are assigned to local decisions (Refer to ***Explicit local decisions based on data*** (on page 88)).

Mapping
A set of rules, describing the translation of one specification into another

Message
A specification of a communication between two MSC instances; the output of the message as well as the input of the message are two MSC events

Model
A semantically closed abstraction of a system

MSC
Message Sequence Chart. MSCs are defined by ITU in [Z120-96, Z120-2000]. May be either basic MSCs or HMSCs.

MSC document
A collection of one or more MSCs

MSC Specification

The set of MSC diagrams describing a system.

PR

Phrase representation

Plain SDL

code generator that produces a minimal SDL model

Scenario

A specific sequence of events that illustrates and specifies behavior by ordering events

SDL

Specification and Description Language, is defined by ITU in standard Z.100

State

A condition or situation during the life of an object during which it satisfies some condition, performs some activity or waits for some events

State machine

A behavior that specifies a sequence of states an object goes through its lifetime in response to events, together with its responses to those events

Synthesis

A deep transformation that produces a (detailed) implementation, conforming to a given (abstract) specification; usually there exists more than one approach to satisfy the specification

System

Behavior, organized to accomplish a specific purpose and specified (externally) by a collection of scenarios that describe its interactions with external actors; a system can structurally decomposed into a collection of subsystems and specified (internally) by a collection of scenarios that extend external scenarios to describe the interactions between subsystems

Textual representation,

Notation, describing the syntax (linear phrase structure) of a specification

Transition

A relationship between two states indicating that an object in the first state will perform certain specified actions and enter the second state when a specified event occurs and specified conditions are satisfied

Type-based SDL

code generator that produces a rich type-based SDL model that extensively uses SDL types to allow evolution of the model without editing the synthesized part.

Index

KLOCwork Corporation

Toll-free telephone: 1-866-556-2967
E-mail: sales@klocwork.com **or**
support@klocwork.com
Website: http://www.klocwork.com

Corporate Headquarters:
1 Antares Drive, Suite 510
Ottawa, Ontario, Canada
K2E 8C4
(613) 224-2277

US Headquarters:
1700 Montgomery Street
Suite 111
San Francisco, CA
94111
(415) 954-7154