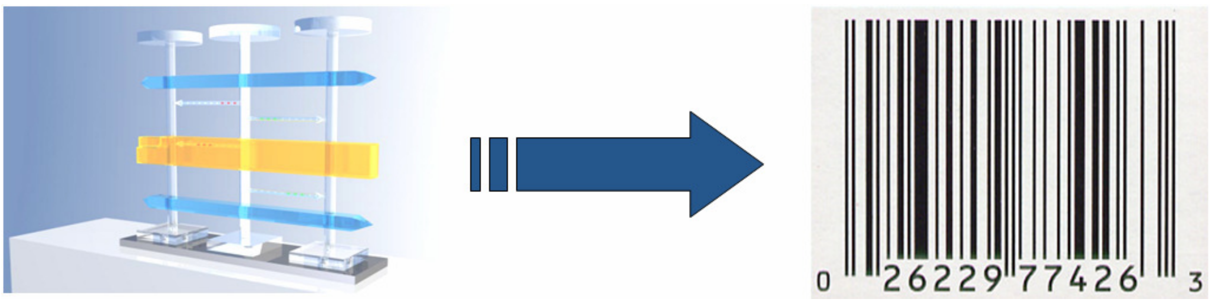
User Manual: MSC to C Test Code Generator



Abstract:

The MSC to C test code generator generates C code for testing from Message Sequence Charts (MSC) models. MSCs have been standardized from the ITU in Z120. In addition to the stimulating test code generation code stubs for testing reactions are generated. The execution of the generated code produces protocols in MSC format, such that testing can be completely model based. Since the C interface to the test object is flexible, the generated test code can be executed in many different settings: Within modelling tools like Matlab/Simulink or Rhapsody, as module test completely on PCs, as integration test using c programmable interface e.g. to CAN bus and directly on the target hardware.

The MSC to C test code generator has been developed by Validas AG, initially supported from the research project mobilSoft, founded by bayerische Staatsregierung. Currently the MSC to C test code generator is a pre-product that may only be used in projects, were it has been methodically introduced and configured by Validas AG.

VALIDAS
validated quality

Contents:

1 Introduction 5

2 The MSC to C Code Generator 6

3 Installation 8

4 Generation Process 10

5 Generation Principles 11

5.1 Testable MSC Subset 11
5.1.1 Supported Constructs and Restrictions 11
5.1.2 Syntactic Well-Formedness 13
5.2 Generation Semantics 14
5.2.1 HMSC 15
5.2.2 MSC 16
5.2.3 Timing 20
5.2.4 Errors 21
5.3 Adaptations 22
5.4 Options 23
5.4.1 Code Generation Options 23
5.4.2 Execution Options 25
5.4.3 Protocol Generation Options 26

6 Code Architecture 27

7 Examples 30

7.1 MiniTimer 30
7.2 FM99 31
7.3 SeqTest 32
7.4 CoregionTest 34
7.5 OptMsg 35

8 Adaptations 37

8.1 Formatter 37
8.2 Types and Return Values 37
8.3 Specification Independent Code 39
8.3.1 main 40
8.3.2 msc2c_types 40
8.3.3 msc2c_time 40
8.3.4 msc2c_debug 40

9 Extensions 46

9.1 MSC Catalogue Generator 46
9.2 Matlab Catalogue Generator 46
9.3 Matlab Integration 47
9.4 MSC Transformations 49

10 Restrictions 51

A. Appendix MPR Grammar 52

Figures:

Figure 1: Overview of the Generation..6
Figure 2: Generation with Function Catalogues..7
Figure 3: MSC to C Setup Program..8
Figure 4: MSC to C Installer..8
Figure 5: MSC to C License Conditions..9
Figure 6: SuT Axis in MSC..16
Figure 7: Detected Errors in Protocol MSC..22
Figure 8: Architecture of the Generated Code..28
Figure 9: Example Generation of Test Code..31
Figure 10: Generation Result of the Example..31
Figure 11: FM99 Example Generation..32
Figure 12: Declaration of a Start Method in the SuT..33
Figure 13: Seq Example: Main Test..33
Figure 14: Seq Example: Checking Stubs..34
Figure 15: Example Optional Message..35
Figure 16: Result of Optional Message Test..36
Figure 17: Changing Return Values of Stubs..39
Figure 18: Integration of MSCs in Matlab/Simulink..47
Figure 19: Properteis of the MSC Driver..48
Figure 20: Transformation Rule..50
Figure 21: Original MSC..50
Figure 22: Transformed MSC..51

1 Introduction

The MSC to C test code generator ("MSC2C") generates ANSI C code from MSCs models. It includes function catalogues for interface specifications, high-level MSCs (HMSC) for system states and MSCs for the test sequences. Many constructs like messages, conditions, actions, timers, references, options and alternatives can be used, however not the complete standard is supported. The restrictions, described in Section 5.1 ensure that the MSCs are unique and can be used for testing. They can be seen as modelling guidelines for message sequence charts.

The manual is structured as follows: An overview of MSC2C is given in Chapter 2. Chapter 3 describes the installation of the code generator and Chapter 4 describes the generation process of the code. The generation principles are explained in Chapter 5. Chapter 6 describes the architecture of the generated code that is the basis for adoptions in Chapter 0 to specific test environments. Examples of the generation are described in Chapter 7. Some limitations are listed in Chapter 9.

2 The MSC to C Code Generator

The MSC to C test code generator generates test code from MSCs. The test code stimulates the subject under test (SuT) and observes the reactions of the SuT.

As Figure 1 shows, the inputs for the generator are MPR files (MSC textual representations) as standardized by the ITU. The generated code mainly consists of the test driver and the stubs. The test driver drives through the test by stimulating the SuT, checking conditions, executing actions and computing the test result. The stubs are called from the SuT and store the passed values for the test driver.

The generated code has to be compiled together with the SuT code. Execution of the test produces a test result, consisting of the number of found errors, an MSC and state coverage of the specification and a test protocol in MPR format.

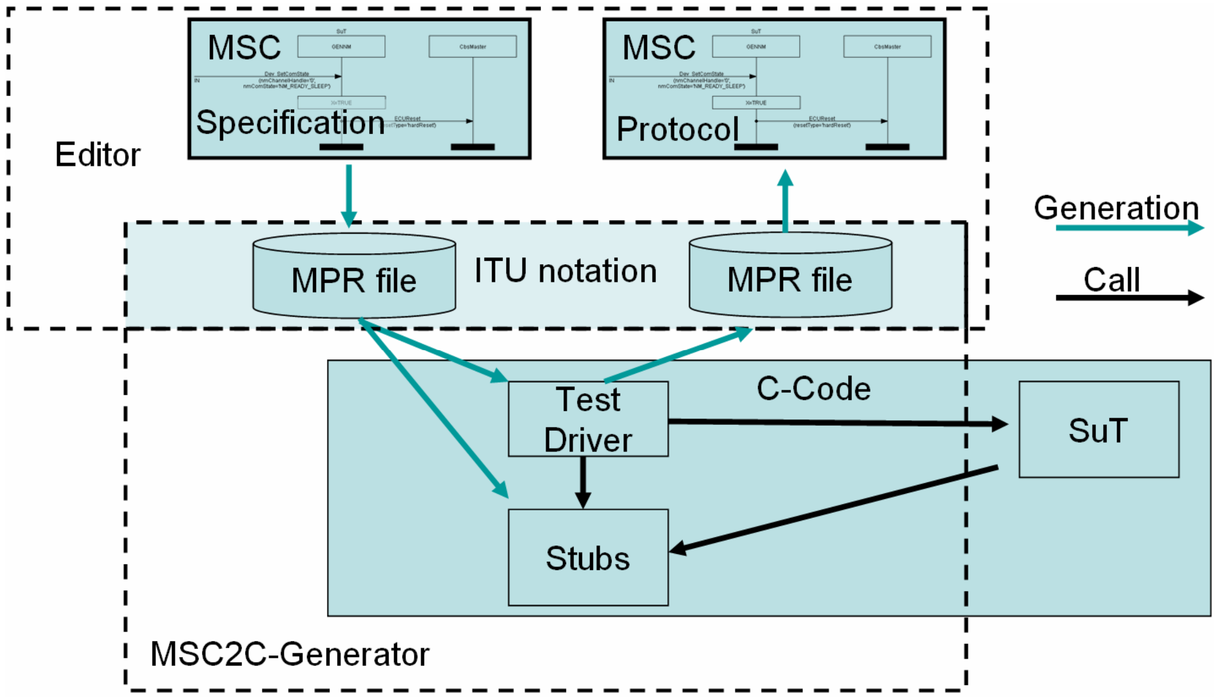


Figure 1: Overview of the Generation

Important for successful test (especially for the compilation) is that the interfaces of the SuT fit to the generated code. Therefore the interface specifications are the basis for the MSCs and have to fit to the SuT. Using simple C code wrappers for the SuT the generated code can test arbitrary objects like embedded targets connected via busses or MATLAB models connected via s-function wrappers, or simple module tests without wrappers.

Since MSCs do not have a possibility to specify interfaces and types the MSC to C generator uses a very general typing that has to be mapped to the types from the SuT, e.g. by **`#define Message_Param1_Type uint8`**. Another more helpful approach is to use so called "function catalogues", which are an XML-representation of the available messages together with their types. Figure 2 shows the extended generation with function catalogues.

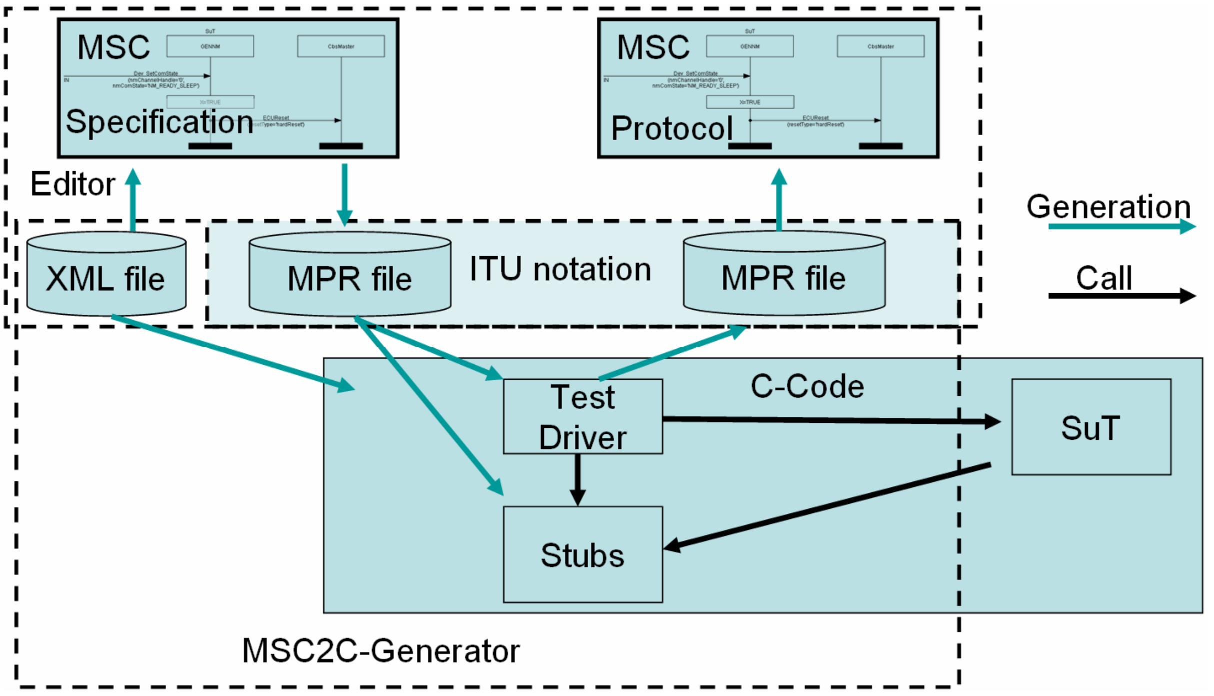


Figure 2: Generation with Function Catalogues

Powerful MSC editors can use the function catalogues to support editing, e.g. by offering pull-down menus for the selection of messages and parameters.

Note that the interface / function catalogue is the most important link between the implementation (SuT) and the test specification (MSC) and should therefore be developed before the specification and the implementation. Using function catalogues in the generation of MSC to C code reduces the necessary work to adopt the interfaces as mentioned above.

3 Installation

MSC to C test code generator requires a java runtime engine (JRE) in the version 1.5 or higher. Installation works on all Window platforms and has been tested under Windows XP and cygwin.

Testing requires to compile and to link the generated C code with an arbitrary ANSI C code compiler or IDE like gcc or visual studio.

The MSC to C test code generator uses a standard installer. By clicking on the setup icon (see Figure 3) the installer (see Figure 4) starts.

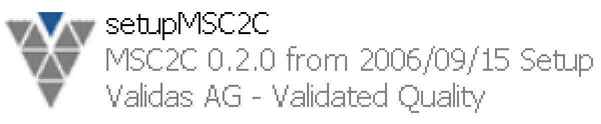
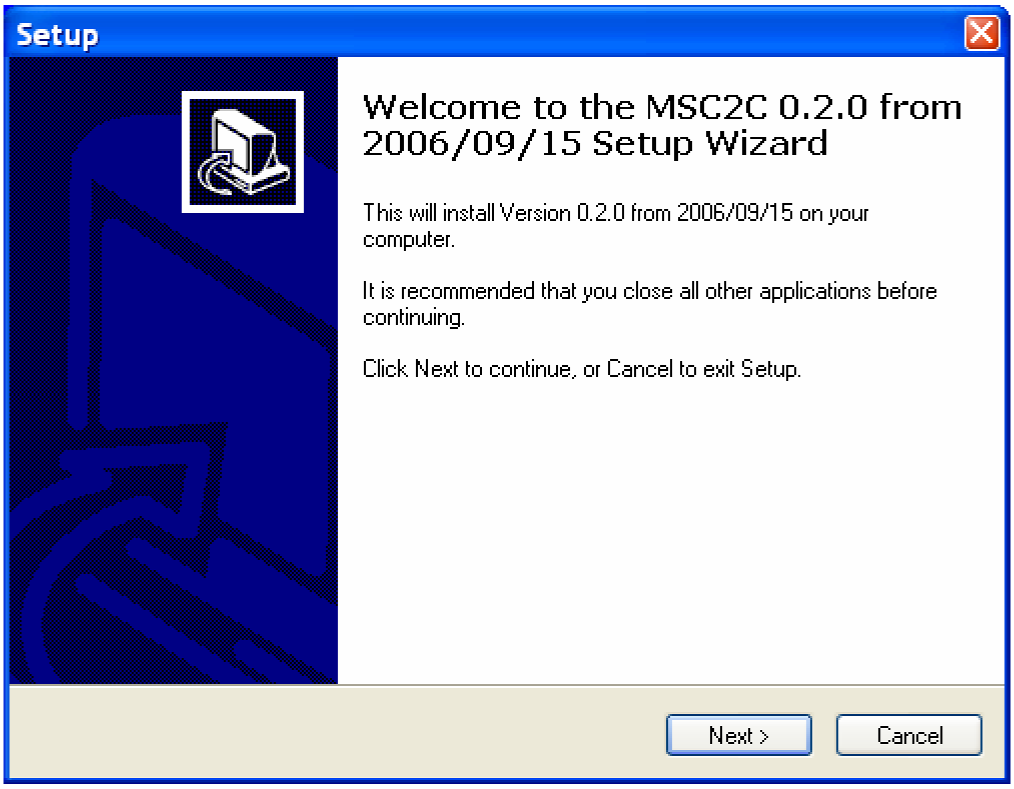

setupMSC2C
MSC2C 0.2.0 from 2006/09/15 Setup
Validas AG - Validated Quality

Figure 3: MSC to C Setup Program



Setup

Welcome to the MSC2C 0.2.0 from 2006/09/15 Setup Wizard

This will install Version 0.2.0 from 2006/09/15 on your computer.

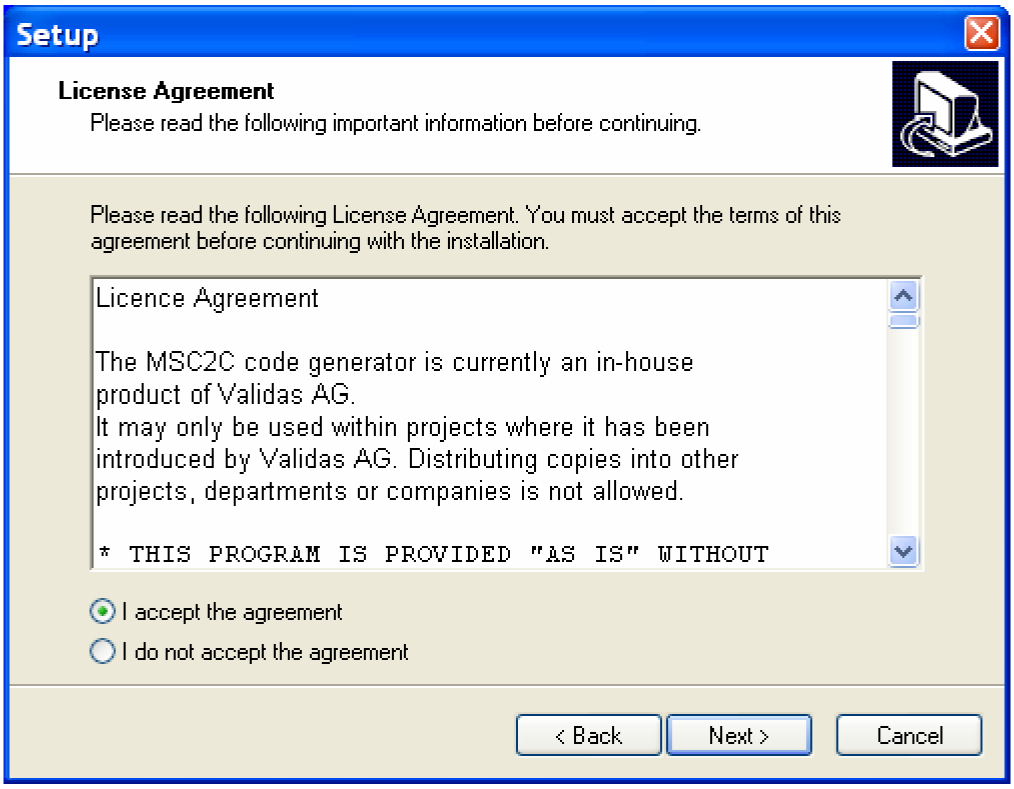
It is recommended that you close all other applications before continuing.

Click Next to continue, or Cancel to exit Setup.

Figure 4: MSC to C Installer

First the user has to accept the licence conditions (see Figure 5), that forbid to copy MSC 2 C test code generator to other projects than those where Validas AG has introduced it.

The next steps require selecting the destination directory (e.g. c:\Program Files\MSC2C), to select the program folder for the Start menu, etc.


Setup
License Agreement
Please read the following important information before continuing.

Please read the following License Agreement. You must accept the terms of this agreement before continuing with the installation.

Licence Agreement

The MSC2C code generator is currently an in-house product of Validas AG.
It may only be used within projects where it has been introduced by Validas AG. Distributing copies into other projects, departments or companies is not allowed.

* THIS PROGRAM IS PROVIDED "AS IS" WITHOUT

[✓ I accept the agreement]
I do not accept the agreement

Figure 5: MSC to C License Conditions

After successful installation the execution path is adopted, such that the following programs can be started from a program shell (like **`cmd`** or cygwin):

- **`msc2c.bat`**: generates the c code from MSCs (see Section 4)
- **`xml2msc.bat`**: generates protocol MSCs from test executions (see Section 4)

Into the start menu program folder a un-installation program for MSC to C is added.

4 Generation Process

The MSC to C code generation is integrated into a testing process for a given SuT (with a C code interface) as follows:

1. Edit the MSCs and store them into an MSC-Document (with several MSCs) MPR-File
2. generate c code using **`msc2c.bat`**
3. compile c code and link it to the SuT (or it's c interface)
4. executing the c code yields
    a. test result (number of detected errors)
    b. test coverage
    c. test protocol in XML format (if selected with the option **`–gen=file.xml`**)
5. generate the protocol MSC (if created in step 4.c) using **`xml2msc.bat`**

If the protocol does not produce the expected results the MSC or the SuT have to be adopted.

The generation process requires some preparation. The main goal of the preparation is to compile the generated code together with the SuT interface. The amount of adoption work depends on the number of functions and arguments in the interface of the SuT. The generated code (see Section 6) consists of test code that shall not be adopted and wrapper code that can be manually adopted.

Note that defaults for the wrapper code are generated together with the test code. After adaptation of the wrapper code this generation has to be switched off with the option **`–nocp`**. Another possibility (which is recommended) is to add the adoptions to the data declarations of the MSC or the function catalogues (see Section 6).

5 Generation Principles

The code generation works for a subset of MSCs. The subset is described in Section 5.1 and can also be considered as modelling guidelines for testable MSCs. The translation semantics are described in Section 5.1.1. The generation options (see Section 5.3) allow controlling some semantic and syntactic aspects of the generation.

5.1 Testable MSC Subset

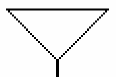
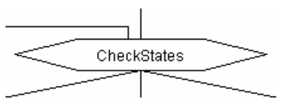
The MSC to C test code generation does not work for the complete MSC standard, but for a subset of testable MSCs. In contrast to MSCs, which can describe arbitrary observable sequences, using MSCs for testing requires to derive concrete input values for driving the tests (for example don't care values (_) in messages can only be used for messages coming from the SuT, but not for stimulation).

The conformity of a MSC to the testable subset is checked before the generation. If the MSC is not testable the generation will not be started and the violations are reported on the output console.

The following subsections describe the separate rule of subset.

5.1.1 Supported Constructs and Restrictions

The following tables (for HMSCs: Table 1 and MSCs: Table 2) show the supported constructs that can be used for modelling.

	The **start symbol** is a triangle. The start symbol marks the start state that is entered without further specified actions, e.g. by starting the software, or connecting the ECU to power. There may be only one start symbol per HMSC.
	A **condition** is depicted with a hexagon. Conditions describe the states of a system. The states are connected using references to MSCs that represent the transitions in the system. Outgoing references are beyond the state, incoming references above the state. Conditions can be setting conditions (with "when") or checking conditions like "when finished". In the later case the conditions have to be set somewhere in the MSCs / HMSCs.
Test(variables 'CHN': 'int8';)	A **parameter** of a HMSC is declared after the name of

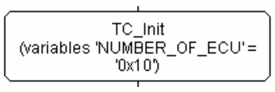
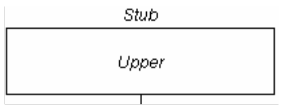
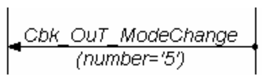
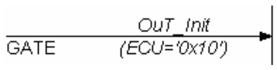
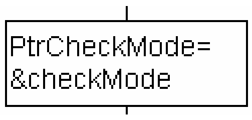
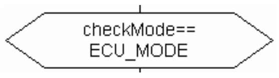
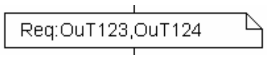
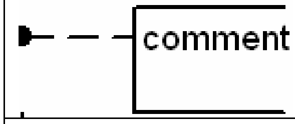
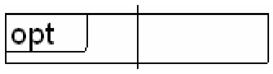
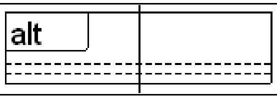
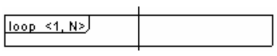
	the HMSC with the keyword ‘variables’. Parameter declarations consist of the name and the type of the parameter.
	An **MSC reference** is depicted as an oval box with the name of the referenced MSC and the instantiations of all parameters of the MSC. A reference in an HMSC must connect two conditions / states. If a reference has several successor states, they must be “when states” or “otherwise”.

Table 1: Supported Constructs in HMSC

	An **axis** in the MSC is depicted a vertical line under a box. The name of the axis is written in the box, the kind of the axis can be written above the box. Axes are used to describe the components in a system.
	A **message** is depicted as an arrow. Messages can be between axes or connect one axis with the environment. The direction of the message (incoming or outgoing from the SuT) depends on the axis, which corresponds to the SuT. Message parameters are appended in brackets.
	A **gate** is an interface of MSCs. Incoming and outgoing messages can use gates.
	An **action** is depicted as a square box. The action that is executed is written in the box. The action shall be executable and might refer to (defined) variables and help functions.
	A **condition** is depicted as a hexagon. The condition that has to be fulfilled is written in the box. The condition shall be executable and evaluate to true or false. It might use (defined) variables and help functions.
	A **text** element is depicted in a box with a folded corner. Text elements are used as comments in MSCs.
	A **comment** annotation can be added to many elements in MSCs.
	An **option** is a box, marked with “opt”. The elements inside the box are optional.
	An **alternative** is a box, marked with “alt”. Dashed lines separate the elements inside the box.
	A **loop** is a box, marked with “loop”. The elements inside the box are repeated. Loops can have a start value and an end value to indicate the number of repetitions. Loops starting with a condition (“while-loops”) are repeated as long as the condition holds.

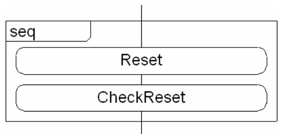
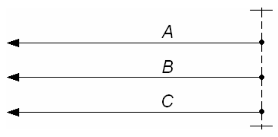
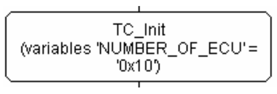
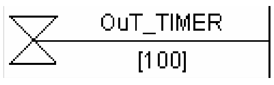
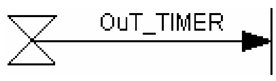
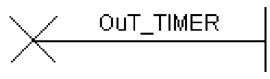
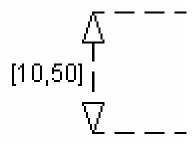
	A **sequence** is a box marked with "seq". All elements inside the box are executed as one sequence. This allows to combine several sequences and to build modular models. The checks (for absence of other messages) at the end of MSCs are suppressed in all sequences within the sequence box and executed at the end of the sequence box.
	A **coregion** is a region of an axis, marked with a dashed line. Elements within coregions are executed concurrently, i.e. the order between the elements is not relevant.
Test(variables 'CHN': 'int8')	A **parameter** of an MSC is declared after the name of the MSC with the keyword 'variables'. Parameter declarations consist of the name and the type of the parameter.
	An **MSC reference** is depicted as an oval box with the name of the referenced MSC and the instantiations of all parameters of the MSC.
	A **start time** action starts a timer. The name of the timer is above the line, the time beyond.
	A **timeout** condition denotes the timeout of a timer. The name of the timer is above the line.
	A **stop time** action stops a running timer. The name of the timer is above the line.
	A **time annotation** denotes the time interval between the two annotated elements The second element has after the minimal time and before the maximal time bound has reached.

Table 2: Supported Constructs in MSCs

5.1.2 Syntactic Well-Formedness

Syntactic well-formedness is checked during parsing of the MPR-Files. The following restrictions are required:

1. correspondence to the ITU grammar (see Appendix A)
2. well-formedness conditions of MSCs that can be checked from the MSC-Editor
    - the referenced MSCs are contained in the document
    - no parameter is missing in the message.
3. correspondence to supported subsets of the grammar (see Appendix A)
4. semantic checks for testability:

- The kinds (condition / action) of the first elements in alternative blocks are identical. The kinds of the modelling elements are:
  - action: action
  - message to SuT: action
  - start/cancel of timer: action
  - condition: condition
  - message from SuT: condition
  - timeout of timer: condition
  - Option: condition
  - Alternative: first elements (all of the same type) in the alternative
  - Loop: action
  - MSC-Reference: kind of first element in the referenced MSC.
- First elements in options must be of the kind condition.
- No message is used in two different gates. If no gates are specified for a message this check is omitted.
- All parameter names in different occurrences of message with the same name are identical.
- No don’t care values are in messages that are sent to the SuT.
- MSC References in HMSC must start and end into a condition node.

The MSC editor ensures many of them by construction, others (like the reference correctness) can be ensured as syntactic checks in the editor.

5.2 Generation Semantics

The generation semantics have been designed for correspondence with the MSC standard. This section describes the realization of the different constructs.

5.2.1 HMSC

HMSCs are realized as state machines using state variables for each condition. For each HMSC in the document following functions are generated:

- **`int <HMSC>()`**: the function that executes the complete test and returns the number of errors found. If the HMSC has parameters these are overtaken into the function.
- **`int <HMSC>_Step()`**: the function that executes one step of the test until the next HMSC state is reached or an MSC waits for the reaction of the SuT.
- **`int get_<HMSC>_State_Coverage()`**: the function that returns the number of (completely) covered states in the HMSC. A state is completely covered, if it has been reached and all outgoing transitions have been covered. The maximal number of states is **`<HMSC>_LAST_STATE_VALUE`**.

For every **condition** in the HMSC a state is generated. In this state a function is executed that selects the test(MSC) to execute the test according to the following heuristic:

- select the executable tests (References to MSCs): MSCs of kind action are always executable, MSCs of kind condition can only be executed if the condition is true. Note that this check can be disabled using the option **`–noguards`**.
- Among the executable tests the test with the least coverage are executed.
- Loops (references to MSCs that do end in the starting state) are preferred to keep the overall test execution short.
- The user can increase the priority of an MSC using the option **–up.**

For every **MSC-Reference** (“transition”) in the HMSC (and all referred MSCs) the call of a test function is generated that executes the test once.

5.2.2 MSC

MSCs are realized as functions with a state. The state corresponds to the position of the tests within the execution. If the state is zero the test is finished.

- **`int <MSC>()`**: the function that executes the test of the MSC and returns the number of errors. If the MSC waits for a condition to become true, the MSC-State is not null. The MSC-states are stored in a field.
- **`BOOL guard_<MSC>()`**: this function is generated for MSCs with condition kind. It returns TRUE if the corresponding test can be executed. The generation of this functions can be disabled using the option **`-noguards`**. Note that the guards check all conditions in the MSC and not only the first.

The translation of an MSC into test code depends on the selected target. One Axis has to be specified as target. This can be done using the axis instance kind "SuT" in the MSC (see Figure 6) or with the generation option **`-sut`**.

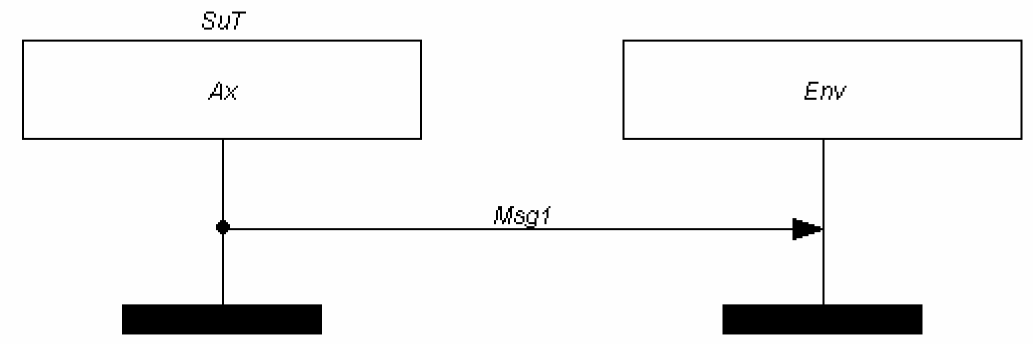


Figure 6: SuT Axis in MSC

The selected SuT determines the semantics of test code generation. The test code is generated for every element in the MSC. The semantics of the elements is described in the following list:

- **Message to SuT**: is translated into a method call to the SuT: All parameters of the message are passed in the order of their specification. The names of the parameters are ignored. If a parameter with name **`return`** is specified then its value is compared with the return value from the SuT.

- **Message from SuT**: is translated into a check if the stub has been called from the SuT. The test driver waits at most the `WAIT_MSG` time for the condition. This time can be configured using the execution option `-wm`. If there is a time annotation on the message (e.g. `[10,100]`), the maximal value (`100`) of the constraint is used as maximal waiting time. If there are more time constraints for the same message, the maximal time which is waited for the message is the maximum of all time constraints (even if the earlier ending constraints might fail, if the messages arrives at the latest possibility). If a message does not arrive within the specified time, an error is reported.
- **Actions**: are directly inserted into the generated test code. If not present, an ";" is appended.
- **Conditions**: are inserted as checks (`if (<cond>)`) into the generated test code. Note that "Named Conditions" of the MSC standard are also supported. A named condition is a condition with a name instead of a boolean term. Named conditions are checked using "when name" and set using name. In the case of setting named conditions a global variable is assigned to true. When checks of named conditions are translated into the check of the variable. To differentiate between Boolean variables and setting named conditions the generation computes if the condition is checked somewhere in the document using when. In this case it is treated as named condition, otherwise as boolean term. The time for waiting for a condition to become true is `WAIT_COND` and can be specified using the execution option `-wc`.
- **Options:** are translated into conditional code. The first element in the option (which is a condition node) is used as condition. All remaining elements in the option are tested if the condition is true. If the first node in an option is an outgoing message, the option will be tested if the message is present after a certain amount of time. This amount depends on the used time annotations are used a

default time (`WAIT_OPT_MSG`) is used. This default time can be configured using the option `-wo`. The semantics of optional message is illustrated in Section 7.5.

- **Alternatives**: have a special semantics depending on the SuT. If the nodes in the alternatives are condition nodes the test driver checks the disjunction of the conditions. If one alternative condition is true, the corresponding alternative is tested. The waiting time for an alternative to become true is `WAIT_ALT`, it can be configured using the option `-wa`. If the alternatives start with messages to the SuT, the alternatives describe several test cases. The choice which test case is executed can be made from the test driver. Therefore a complete test of "environment alternatives" requires testing all alternatives. In this case the generated test function has an additional parameter (`int iAlt`) and the coverage of the test is only reached if all alternatives are tested. If several alternatives are in one MSC the overall number of alternatives of the MSC is the maximum of the alternatives.
- **Seq**: A sequence causes the contained elements to be executed as one sequence with one absence check at the end of the Seq block (see the example in Section 7.3)
- **Loops**: A loop is translated into a while loop, with the first condition as guard. An index variable is used for counting the repetitions of the loop. At the end of the loop the specified repetitions are compared with the executed repetitions.
- **Timers**: are realized using timer structures that store the starting time. If timers are checked for timeout, the elapsed time is compared with the specified time. Similar to conditions and messages there is a wait time for timeouts. The time for waiting for a timeout to arrive is `WAIT_TIMEOUT_TIME` and can be specified using the execution option `-wto`.
- **Time annotations**: can be attached between two so-called timeable events (see ITU Z 120). For example messages and

conditions. The test code generator supports two kinds of time annotations: exact time annotations and interval time annotations. Exact time annotations are treated as intervals with identical upper and lower bound. Before the execution of the first timeable event a timer is started. The second timeable node is executed after the lower bound of time. If this is not reached during execution the test waits until the lower bound is reached. The upper bound of time annotations is checked after the execution of the timeable event. Note that this semantics does not interrupt the test after the maximal amount of time, but issues an error if the bound is violated.

- **Coregions**: describe parallel actions. In the case of messages from the SuT their order is not checked. In case of other actions all possible permutations are tested (like environment alternatives). The coregion generation can be seen by the example in Section 7.4.

At the end of an MSC the SuT is checked for silence, i.e. if the SuT has sent unspecified messages to the SuT. The default behaviour is that all stubs are checked for silence. The messages can be partitioned into groups using gates. For every MSC the gates that are checked for silence can be specified with a comment element with the following form checkgate = <value>

Possible values for checkgate are:

- all: all gates (with all messages from SuT). This is the default setting
- used: all gates that are used in the MSC
- gate terms like `g1+g2` or `g1,g2` or `all-g2`
- single messages can be added to the gates, like `used-cyclicM1`

The generation of the silence check at the end of each MSC test can be avoided by using a sequence box, which denotes that the element in the box (mostly references to other MSCs) are executed within one sequence, not separated by silence checks. The silence check is executed at the end of the Seq block.

5.2.3 Timing

The MSC standard allows to use time annotations and timers. Their value are usually interpreted in milliseconds. Since timing depends on the execution of MSC, the test code generator cannot ensure correct timing, however it has a flexible timing concept that can be adopted easily to meet the timing requirements.

The default timing concept is based on a counter (stCurStep) that is incremented every time the test code is called. Hence calling the test code every millisecond suffices to ensure correct timing.

Working with devices that provide access to hardware timers can also be used for timing. In this case the time module `msc2c_time` (see Section 6) has to be adopted manually and it's generation must be prohibited using the option **–nocptime** (se Section 5.4.1). The measured real time has to be set from the user into the timing module (from where it is used during the tests). Setting the real time can be done with a function, e.g.

```
/* help variable for getTimer */
static TIME_TYPE    ttTime = 0;

/** this method returns the current time as set from the user */
TIME_TYPE getTime() {
    return ttTime;
}
/** this method must be used to set the time */
void setTime(TIME_TYPE tNew) {
      ttTime=tNew;
}
```

Using such an implementation requires that the time is set correctly from the user, e.g. using the time stamp of a message receive event (`xlEvent`) from the vector CAN card:

```
        setTime(xlEvent.timeStamp/1000000); /* convert ns to ms */
```

Note that working with such time concepts requires to provide the time constantly (and not only if messages are received).

5.2.4 Errors

The detected errors are counted using the global variable **`iAllTestErrors`** (declared in **`msc2c_debug`** module). This variable is statically initialized and never reset, such that the user has to reset it, when restarting the tests manually. In addition the errors occurred within one MSCs are counted and reported at the end of every MSC.

Every reported error has a error kind (indicated by a number). The available error kinds are described in the type Errors as declared in **`msc2c_debug.h`**. The following error kinds are defined:

- `OverallTestTimeout,`
- `ConditionViolated,`
- `UnexpectedMessage,`
- `MessageMissing,`
- `WrongMessageSequenceEarly,`
- `WrongMessageSequenceLate,`
- `WrongMessageParameterValue,`
- `WrongReturnParameterValue,`
- `WrongLoopRepetitionNumber,`
- `TimeAnnotationViolated,`
- `NoTimeoutOfTimer,`
- `WrongMSC2CConfiguration`

When MSC protocols are generated the name of MSCs with errors are prefixed with an "Error" tag. A typical example for an MSCs with errors can be found in Figure 7. It shows the tagged name of the protocol MSC (Error_At_11_21_21.970_MTest1) together with three errors. The kinds of the messages can be found in brackts after the erron number, however the textual description is equivalent to the error kind. Note that unexpected messages are not displayed as messages, since they have no model correspondence in the specification. Furthermore Figure 7 shows that "two errors" seem to have the same number, however the second error is just a description summarizing the two previous errors, and therefore has no extra number

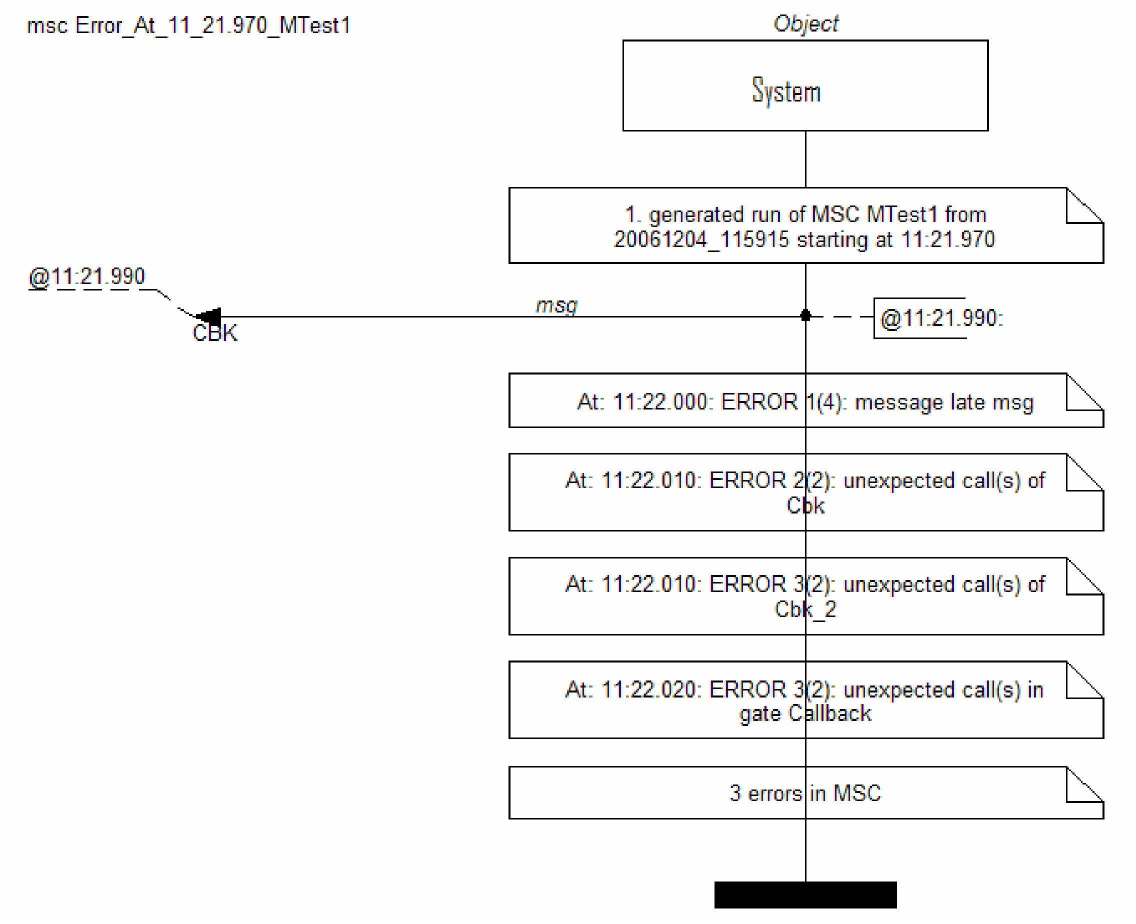

msc Error_At_11_21.970_MTest1
Object
System
1. generated run of MSC MTest1 from 20061204_115915 starting at 11:21.970
@11:21.990
msg
CBK
@11:21.990:
At: 11:22.000: ERROR 1(4): message late msg
At: 11:22.010: ERROR 2(2): unexpected call(s) of Cbk
At: 11:22.010: ERROR 3(2): unexpected call(s) of Cbk_2
At: 11:22.020: ERROR 3(2): unexpected call(s) in gate Callback
3 errors in MSC

Figure 7: Detected Errors in Protocol MSC

5.3 Adaptations

The semantics of the generated code can be tailored for using it in different contexts. The generated code has several specification independent modules that can be adopted by the user. In order to provide an executable test code (and a starting basis for the adaptation) defaults are generated (mainly by copying) for these modules. If these defaults have been adopted, the generation of the default files has to be disabled, for example with the option **–nocp**, otherwise MSC2C overwrites the changes. Note that adaptations can be misused for specification issues, e.g. .to declare a variable that is needed in the test specifications; however this shall be avoided, e.g. by declaring the variable in the MSC document. The same holds for types that shall be defined / included from the function catalogue.

This section provides the signatures for the adoptable functions. More details can be found in Section 8.3.

5.4 Options

There are three kinds of options that can be specified: First kind of options are options for the MSC to C test code generator; second kind of options are options that can be specified for the execution of the generated (and compiled) test. Both programs support the option –help to display the available options. The default values for the options are specified in [square brackets], where [] means that option is disabled by default. Finallys the protocol generation can also use some options.

5.4.1 Code Generation Options

The following options can be given to the MSC to C test code generator (Note that the MPR-File(s) and the destination directory must be specified after the options):

- -**nocp** []: do not copy the specification independent files (`main.c` and `msc2c*`). This allows to manually adopt them to project specific settings.
    - **–nocpmain** []: do not copy `main.c`
    - **–nocptypes** []: do not copy `msc2c_types.h`
    - **–nocpuserdefs** []: do not copy `msc2c_userdefs.h`
    - **–nocptime** []: do not copy `msc2c_time.c` and `msc2c_time.h`
    - **–nocpdebug** []: do not copy `msc2c_debug.c` and `msc2c_debug.h`
- -**noformat** []: the formatting of the generated code (see Section 8.1) is disabled.
- -**nouserprio** []: the generation of the user prio methods is disabled.
- -**noguards** []: the guards, which determine the choice of the test to execute in a given HMSC state are ignored in the MSC selection heuristics. This reduces the sophisticated strategy to a simple coverage guide, i.e. the MSC with the least coverage (number of executions) is executed. The coverage is evaluated transition based.

i.e. an MSC, which has only been executed in another state, has the same coverage as an unexecuted MSC. In the case of several equally covered MSCs in a state, the order of the MSCs in the document determines the execution. Note that this strategy can lead to inefficient executions, since instead of waiting in the HMSC state until one MSC is executable an arbitrary MSC is started.

- -**nomsgorder** []: the order of messages from the SuT is relevant in the semantics of MSCs. However in many test scenarios (like software module test) the order of messages is not relevant. This can be specified using a coregion in MSCs, however to simplify specifications the coregions can be omitted using this option.
- **-wc [0]:** default value for `WAIT_COND` (see Section 5.2.2).
- **-wm [100]:** default value for `WAIT_MSG` (see Section 5.2.2).
- **-wo [50]:** default value for `WAIT_OPT_MSG` (see Section 5.2.2).
- **-wa [200]:** default value for `WAIT_ALT` (see Section 5.2.2).
- **-wto [10000]:** default value for `WAIT_TIMEOUT_TIME` (see Section 5.2.2)
- **-ext []:** use the extension catalog for generation, especially the type definitions of the DataLanguage section in the catalogue are inserted into the generated `TEST_types.h` file. The extension catalog can also be used for the generation of enumerations or defines (see option –cataloggen)
- **-cataloggen [none]:** can have the values **enum**, **define** and **none**. None does not generate type definitions (except the user defined ones), Enum generates an enum type for each enumeration in the catalogue, define just generates #define commands to define the used values.
- **-valueprefix [""]:** the prefix for the generation of values from the catalogue. This option has effect if a catalogue is specified and the value of cataloggen is define or enum.

- **-typeprefix [“”]:** the prefix for the generation of type names from the catalogue. This option has effect if a catalogue is specified and the value of cataloggen is define or enum.
- **-sut [“SuT”]:** The name of the SuT axis or instance kind.
- **-callprefix [“”]:** the prefix of each called function in the SuT
- **-stubprefix [“”]:** the prefix for the generation of stubs. The stubprefix option allows to use the generated test code integrated with the original code by separating the name space. In the case where an original function is called, the call of the (renamed) stub has to been inserted into the original stub, otherwise the test driver is not notified from reactions of the SuT. By defining the macros CALL_REAL_<function> a call from the stub to a real function can be inserted. Note that the real function cannot have the same name as the stub.
- **-module [“<File>”]:** the file names of the generated code (Usually the name of the MPR file (without the .mpr suffix) is used.
- **-allActions [“false”]:** If all actions=true all actions in the MSCs are generated into the test code (not only those on the SuT).
- **-allConditions [“false”]:** If all conditions=true all conditions in the MSCs are generated into the test code (not only those on the SuT).

5.4.2 Execution Options

The following options can be given to the generated test code (if the default main file is used for argument processing).

- **-d / -D**: enables all debugging messages (default=off)
- **-s=N** starts debugging at step N
- **-e=N** ends debugging at step N
- **-t** : enables all model tracing commands (default=off)
- **-tms** : enables tracing of msc states (default=off)
- **-tp** : trace pending MSCs (in HMSC) (default=off)
- **-dw** : debugging of waiting messages (default=off)
- **-tm=<MSC>**: enables MSC tracing for MSC

- **-gen=<File.xml>**: starts generation of protocol information into the specified file. The file has a simple xml format and is used for the generation of MSC protocols
- **-p <state>_<MSC>**: increases the priority of MSC in State
- **-wm=<value>**: sets the waiting time (in steps) the system waits for the presence of a message
- **-wt=<value>**: sets the waiting time (in steps) the system waits for the presence of a timeout
- **-i** : ignore test errors, exit with 0 (default=off)

5.4.3 Protocol Generation Options

The xml2msc protocol generator requires the original MSC as a reference and the protocol file that has been generated using the option **`–gen`**. In addition there can be a function catalogue (at first place in arguments). If this is present, the order of parameters is adopted to the given function catalogue. Furthermore an option **`–realtime`** can be given that formats the time annotations instead of the time ticks in a realtime format, e.g. At_123456 is formatted as At_02:03.456.

The complete call of xml2msc is:

```
xml2msc EXT.xml MSC.MPR log.xml –realtime
```

6 Code Architecture

The generated code from a file TEST.mpr consists of the following files (The name prefix TEST can be configured to any other value using the option **`–module`**). The architecture consists of MSC specific files, that depend on the test specification (starting with TEST) and of environment specific files (starting with msc2c_). MSC2C generates defaults for the test environment files. Adoptions for the test environment should be made either in the msc2c_*-files (preferably in msc2c_userdefs.h) or in the test specification (consisting of the MPR-File and the data declaration in the function catalogue).

- **`TEST.c`** / **`TEST.h`**: test driver code (and declarations) from the MSC
- **`TEST_stubs.c`** / **`TEST_stubs.h`**: test stub code for the MSC
- **`TEST_model.c`** / **`TEST_model.h`**: model related information
- **`TEST_types.h`**: default types definitions and function catalogue
- **`main.c`**: main program for starting the test
- **`msc2c_debug.c`** / **`msc2c_debug.h`**: debugging routines and generation of protocols
- **`msc2c_userdefs.h`**: user definitions, based on the types (and required in the test specifications. Note: a default for this file is generated from MSC2C to have a complete compiling code, however, when changed from the user generation must be suppressed using he option **–nocpuserdefs**.
- **`msc2c_time.c`** / **`msc2c_time.h`**: declarations of time functions
- **`msc2c_types.h`**: default types for generated code

The architecture (include structure) of the code is depicted Figure 8. It shows the dependencies between the modules.

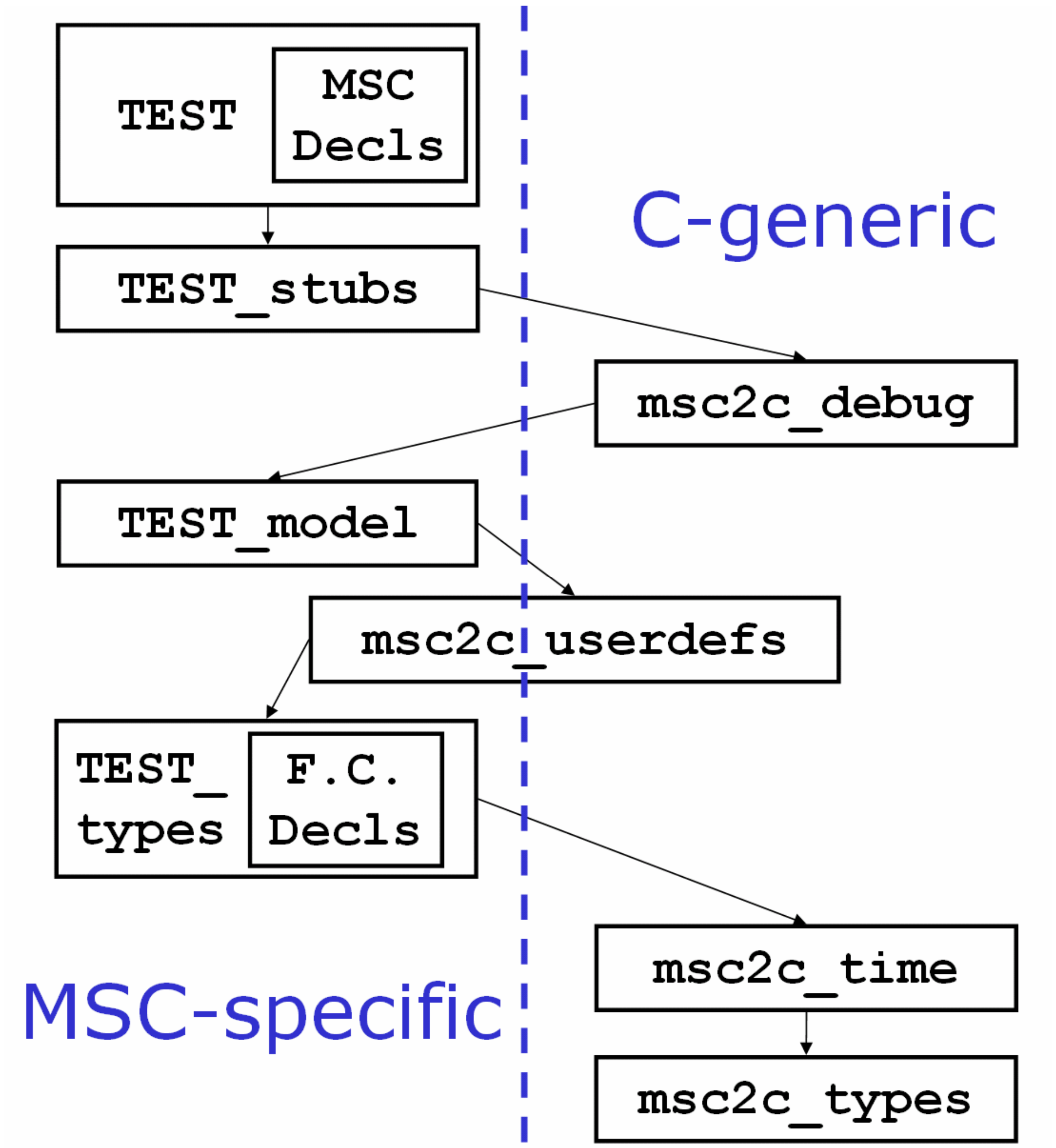


Figure 8: Architecture of the Generated Code

The `MSC Decls` and the `F.C. Decls` are the data declarations that the user can define in the MSC document and the function catalogue. While `MSC Decls` should mainly contain test variables (and types), `F.C.Decls` should contain all types, especially those required for the communication with the SuT, which is in **`TEST_stubs`** (the callback functions, called from the SuT) and **`TEST`** (the calling functions, called from the test driver).

Note that the default types that are generated in **`TEST_types`** if no other types are defined (using **`#define`**) can be avoided by defining them in

`msc2c_types.h`, which is included in **`TEST`** and **`TEST_stubs`** before **`TEST_types`** is included. If a function catalogue is used the adaptations can also be made in the DataDeclarations section of the catalogue.

7 Examples

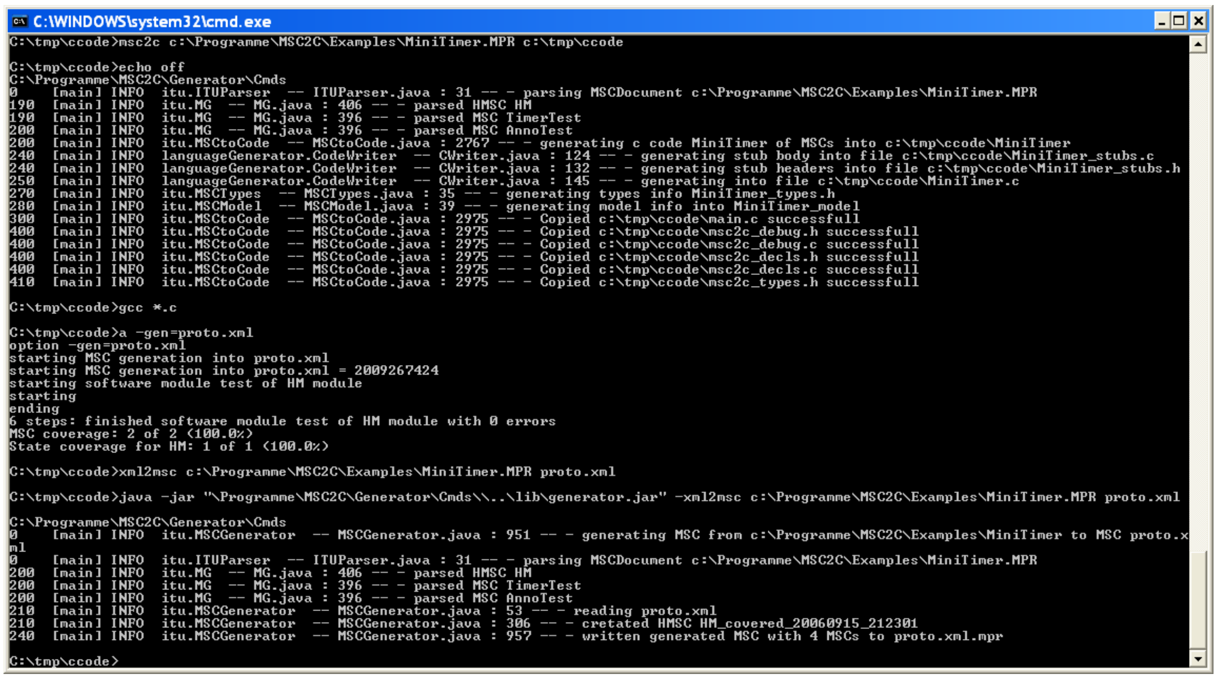
There are several examples included in the distribution of MSC to C test code generator. The examples are located in the subdirectory **`Examples`**.

7.1 MiniTimer

The example **`MiniTimer`** is a very simple test example dealing with a time annotation and a timer. It has no interface to a SuT and hence does not require a function catalogue, a SuT code or any manual adoptions. From **`MiniTimer.MPR`** a complete compiling code can be generated as follows:

1. create a directory for the code, e.g. by:
   ```
   md c:\tmp\ccode
   cd c:\tmp\ccode
   ```
2. generate code by:
   ```
   msc2c c:\Programme\MSC2C\Examples\MiniTimer.MPR c:\tmp\ccode
   ```
3. compile the generated code by:
   ```
   gcc *.c
   ```
4. execute the generated code by:
   ```
   a -gen=proto.xml
   ```
5. generate the protocol MSC by:
   ```
   xml2msc c:\Programme\MSC2C\Examples\MiniTimer.MPR proto.xml
   ```

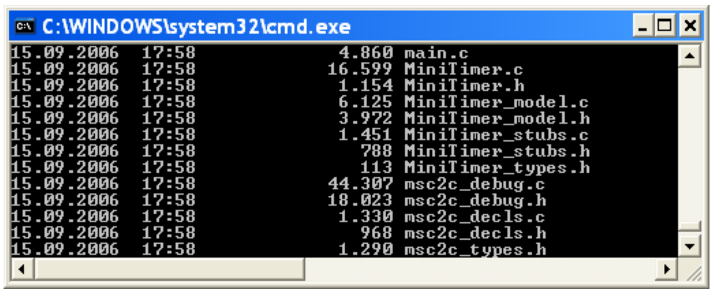
The resulting outputs should look similar to those displayed in Figure 9.



```
C:\tmp\ccode>msc2c c:\Programme\MSC2C\Examples\MiniTimer.MPR c:\tmp\ccode

C:\tmp\ccode>echo off
C:\Programme\MSC2C\Generator\Cmds
0    [main] INFO  itu.ITUParser  -- ITUParser.java : 31 -- - parsing MSCDocument c:\Programme\MSC2C\Examples\MiniTimer.MPR
190  [main] INFO  itu.MG  -- MG.java : 406 -- - parsed HMSC HM
190  [main] INFO  itu.MG  -- MG.java : 396 -- - parsed MSC TimerTest
200  [main] INFO  itu.MG  -- MG.java : 396 -- - parsed MSC AnnoTest
200  [main] INFO  itu.MSCtoCode  -- MSCtoCode.java : 2767 -- - generating c code MiniTimer of MSCs into c:\tmp\ccode\MiniTimer
240  [main] INFO  languageGenerator.CodeWriter  -- CWriter.java : 124 -- - generating stub body into file c:\tmp\ccode\MiniTimer_stubs.c
240  [main] INFO  languageGenerator.CodeWriter  -- CWriter.java : 132 -- - generating stub headers into file c:\tmp\ccode\MiniTimer_stubs.h
250  [main] INFO  languageGenerator.CodeWriter  -- CWriter.java : 145 -- - generating into file c:\tmp\ccode\MiniTimer.c
270  [main] INFO  itu.MSCTypes  -- MSCTypes.java : 35 -- - generating types info MiniTimer_types.h
280  [main] INFO  itu.MSCModel  -- MSCModel.java : 39 -- - generating model info into MiniTimer_model
300  [main] INFO  itu.MSCtoCode  -- MSCtoCode.java : 2975 -- - Copied c:\tmp\ccode\main.c successfull
400  [main] INFO  itu.MSCtoCode  -- MSCtoCode.java : 2975 -- - Copied c:\tmp\ccode\msc2c_debug.h successfull
400  [main] INFO  itu.MSCtoCode  -- MSCtoCode.java : 2975 -- - Copied c:\tmp\ccode\msc2c_debug.c successfull
400  [main] INFO  itu.MSCtoCode  -- MSCtoCode.java : 2975 -- - Copied c:\tmp\ccode\msc2c_decls.h successfull
400  [main] INFO  itu.MSCtoCode  -- MSCtoCode.java : 2975 -- - Copied c:\tmp\ccode\msc2c_decls.c successfull
410  [main] INFO  itu.MSCtoCode  -- MSCtoCode.java : 2975 -- - Copied c:\tmp\ccode\msc2c_types.h successfull

C:\tmp\ccode>gcc *.c

C:\tmp\ccode>a -gen=proto.xml
option -gen=proto.xml
starting MSC generation into proto.xml
starting MSC generation into proto.xml = 2009267424
starting software module test of HM module
starting
ending
6 steps: finished software module test of HM module with 0 errors
MSC coverage: 2 of 2 (100.0%)
State coverage for HM: 1 of 1 (100.0%)

C:\tmp\ccode>xml2msc c:\Programme\MSC2C\Examples\MiniTimer.MPR proto.xml

C:\tmp\ccode>java -jar "\Programme\MSC2C\Generator\Cmds\\..\lib\generator.jar" -xml2msc c:\Programme\MSC2C\Examples\MiniTimer.MPR proto.xml

C:\Programme\MSC2C\Generator\Cmds
0    [main] INFO  itu.MSCGenerator  -- MSCGenerator.java : 951 -- - generating MSC from c:\Programme\MSC2C\Examples\MiniTimer to MSC proto.x
ml
0    [main] INFO  itu.ITUParser  -- ITUParser.java : 31 -- - parsing MSCDocument c:\Programme\MSC2C\Examples\MiniTimer.MPR
200  [main] INFO  itu.MG  -- MG.java : 406 -- - parsed HMSC HM
200  [main] INFO  itu.MG  -- MG.java : 396 -- - parsed MSC TimerTest
200  [main] INFO  itu.MG  -- MG.java : 396 -- - parsed MSC AnnoTest
210  [main] INFO  itu.MSCGenerator  -- MSCGenerator.java : 53 -- - reading proto.xml
210  [main] INFO  itu.MSCGenerator  -- MSCGenerator.java : 306 -- - cretated HMSC HM_covered_20060915_212301
240  [main] INFO  itu.MSCGenerator  -- MSCGenerator.java : 957 -- - written generated MSC with 4 MSCs to proto.xml.mpr

C:\tmp\ccode>
```

Figure 9: Example Generation of Test Code

The generation results are in the directory `c:\tmp\ccode` (see Figure 10).



```
15.09.2006  17:58             4.860 main.c
15.09.2006  17:58            16.599 MiniTimer.c
15.09.2006  17:58             1.154 MiniTimer.h
15.09.2006  17:58             6.125 MiniTimer_model.c
15.09.2006  17:58             3.972 MiniTimer_model.h
15.09.2006  17:58             1.451 MiniTimer_stubs.c
15.09.2006  17:58               788 MiniTimer_stubs.h
15.09.2006  17:58               113 MiniTimer_types.h
15.09.2006  17:58            44.307 msc2c_debug.c
15.09.2006  17:58            18.023 msc2c_debug.h
15.09.2006  17:58             1.330 msc2c_decls.c
15.09.2006  17:58               968 msc2c_decls.h
15.09.2006  17:58             1.290 msc2c_types.h
```

Figure 10: Generation Result of the Example

The result of the xml2msc protocol generation is in the file `proto.xml.mpr`.

7.2 FM99

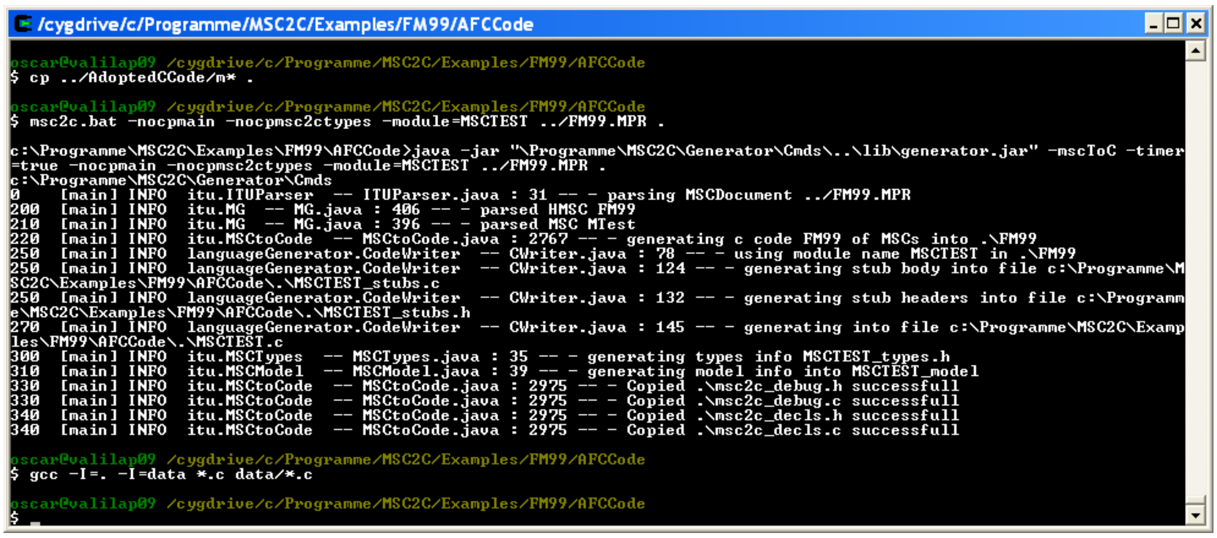
The example `FM99` is a test for an AutoFOCUS model, from which c code has been generated. The test is quite simple, however the interaction between the model and the test code is not by message call, but by global variables. This requires changing the main routine, such that the model and the test are interleaved and the outputs from the model are (manually) notified to the test code. This is done in a manually adopted `main.c` file. Furthermore complex data types are used, such that the

VALUE_TYPE is not the default **`uint8`**, but **`char*`**. Therefore the basic types (**`msc2c_types.h`**) are also manually adopted.

Generating the code would overwrite the adopted files, therefore regeneration of these files is suppressed using the options **`-nocpmain`** and **`-nocpmsc2ctypes`**.

Using the Example directory of the MSC2C distribution the generation works as follows (see Figure 11):

1. **`cd c:/Programme/MSC2C/Examples/FM99/AFCCode`**
2. copy manually adopted files by: **`cp ../AdoptedCCode/m* .`**
3. generate code with: **`msc2c -nocpmain -nocpmsc2ctypes -module=MSCTEST ../FM99.MPR`** . The option -module=MSCTEST is required, since the manual adopted code uses MSCTEST as file prefix instead of FM99.
4. compile the code (note that there is a subdirectory to include) with: **`gcc -I=. -I=data *.c data/*.c`**



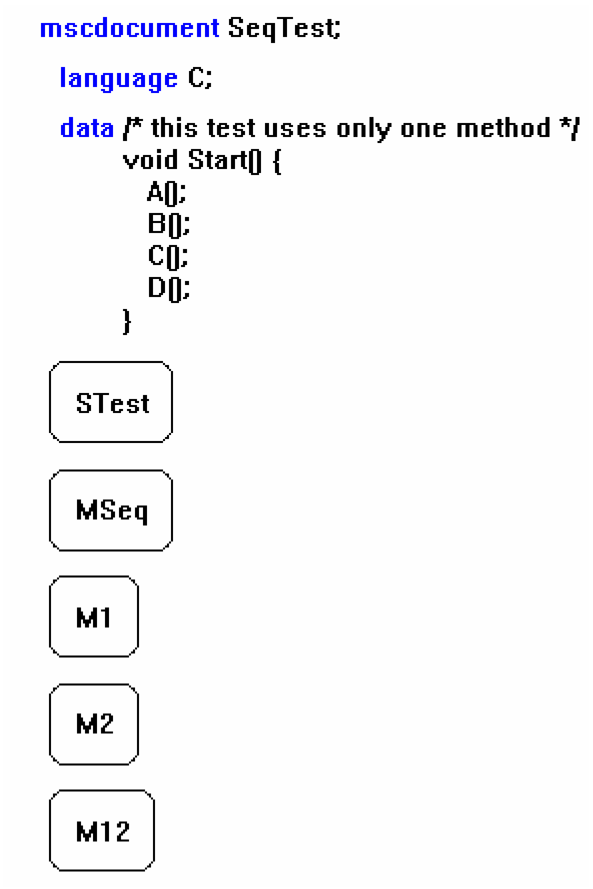
/cygdrive/c/Programme/MSC2C/Examples/FM99/AFCCode

```
oscar@valilap09 /cygdrive/c/Programme/MSC2C/Examples/FM99/AFCCode
$ cp ../AdoptedCCode/m* .

oscar@valilap09 /cygdrive/c/Programme/MSC2C/Examples/FM99/AFCCode
$ msc2c.bat -nocpmain -nocpmsc2ctypes -module=MSCTEST ../FM99.MPR .

c:\Programme\MSC2C\Examples\FM99\AFCCode>java -jar "\Programme\MSC2C\Generator\Cmds\..\lib\generator.jar" -mscToC -timer
=true -nocpmain -nocpmsc2ctypes -module=MSCTEST ../FM99.MPR .
c:\Programme\MSC2C\Generator\Cmds
0    [main] INFO  itu.ITUParser  -- ITUParser.java : 31 -- - parsing MSCDocument ../FM99.MPR
200  [main] INFO  itu.MG  -- MG.java : 406 -- - parsed HMSC FM99
210  [main] INFO  itu.MG  -- MG.java : 396 -- - parsed MSC MTest
220  [main] INFO  itu.MSCtoCode  -- MSCtoCode.java : 2767 -- - generating c code FM99 of MSCs into .\FM99
250  [main] INFO  languageGenerator.CodeWriter  -- CWriter.java : 78 -- - using module name MSCTEST in .\FM99
250  [main] INFO  languageGenerator.CodeWriter  -- CWriter.java : 124 -- - generating stub body into file c:\Programme\M
SC2C\Examples\FM99\AFCCode\.\MSCTEST_stubs.c
250  [main] INFO  languageGenerator.CodeWriter  -- CWriter.java : 132 -- - generating stub headers into file c:\Programm
e\MSC2C\Examples\FM99\AFCCode\.\MSCTEST_stubs.h
270  [main] INFO  languageGenerator.CodeWriter  -- CWriter.java : 145 -- - generating into file c:\Programme\MSC2C\Examp
les\FM99\AFCCode\.\MSCTEST.c
300  [main] INFO  itu.MSCTypes  -- MSCTypes.java : 35 -- - generating types info MSCTEST_types.h
310  [main] INFO  itu.MSCModel  -- MSCModel.java : 39 -- - generating model info into MSCTEST_model
330  [main] INFO  itu.MSCtoCode  -- MSCtoCode.java : 2975 -- - Copied .\msc2c_debug.h successfull
330  [main] INFO  itu.MSCtoCode  -- MSCtoCode.java : 2975 -- - Copied .\msc2c_debug.c successfull
340  [main] INFO  itu.MSCtoCode  -- MSCtoCode.java : 2975 -- - Copied .\msc2c_decls.h successfull
340  [main] INFO  itu.MSCtoCode  -- MSCtoCode.java : 2975 -- - Copied .\msc2c_decls.c successfull

oscar@valilap09 /cygdrive/c/Programme/MSC2C/Examples/FM99/AFCCode
$ gcc -I=. -I=data *.c data/*.c

oscar@valilap09 /cygdrive/c/Programme/MSC2C/Examples/FM99/AFCCode
$ _
```

Figure 11: FM99 Example Generation

7.3 SeqTest

The example **`SeqTest`** shows the semantics of the sequence operation. To demonstrate this the SuT has a method **`start()`**, which calls (the stubs) **`A()`**, **`B()`**, **`C()`** and **`D()`**. This method is provided in the DataLanguage field of the document (see Figure).



```
mscdocument SeqTest;
 language C;
 data /* this test uses only one method */
       void Start() {
         A();
         B();
         C();
         D();
       }
```

Figure 12: Declaration of a Start Method in the SuT

The main test is described in the MSC in Figure 13.

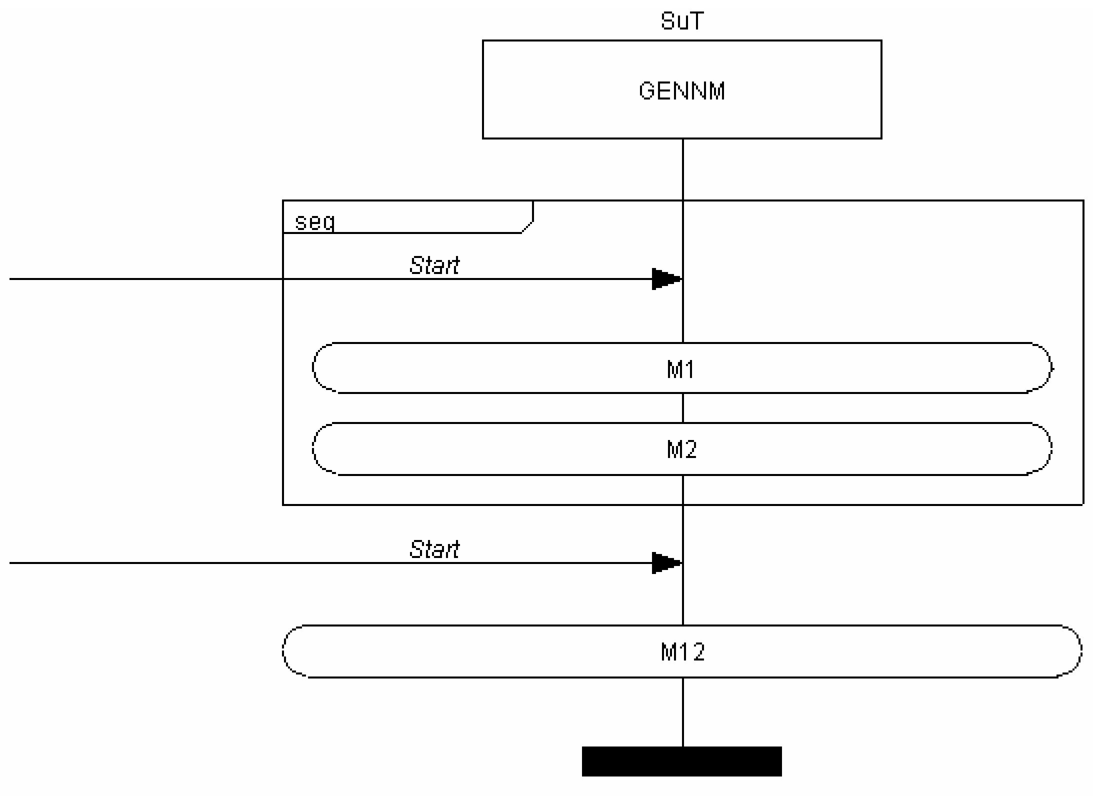


Figure 13: Seq Example: Main Test

It calls start two times and checks whether the stubs have been called. The first check is modular and split into two MSCs **`M1`** checks for the presence of **`A`** and **`B`** and **`M2`** which checks for **`C`** and **`D`** (see Figure 14). The second check (**`M12`**) checks for the presence of all stubs. Mostly the modular approach is preferred, however without the **`Seq`** box in the main MSC the test of **`M1`** would detect the unexpected calls **`C`** and **`D`** and the test of M2 would fail since the Messages **`C`** and **`D`** have been "consumed" from test M1. Therefore using a **`Seq`** operator in this interpretation of MSCs can be useful to combine sequences.

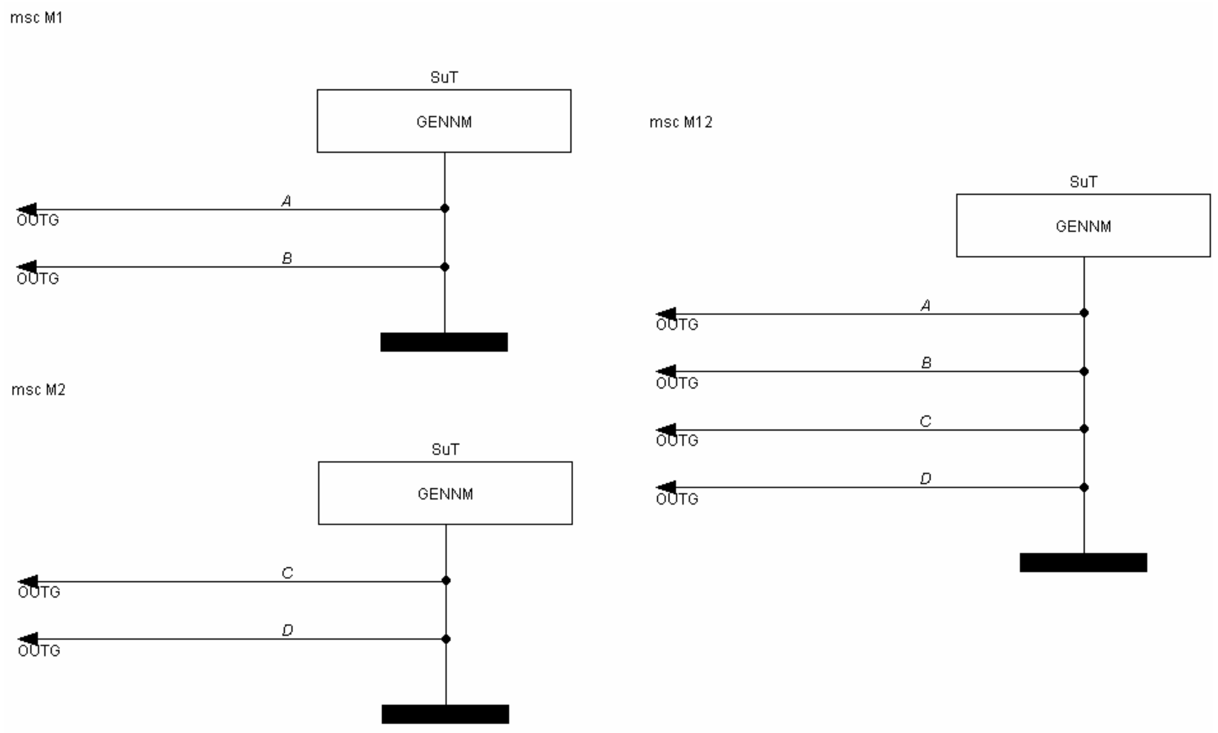


Figure 14: Seq Example: Checking Stubs

7.4 CoregionTest

The example **`CoregionTest.MPR`** (in the examples directory of the distribution) is an example that shows the different usages of Coregions within one test, depending on the kind of the actions inside the coregion. If the kind of the elements are incoming actions, their order is permuted in every possible order (see MSCs **`Permute`** and **`PermuteMany`**). In the other cases, for example the MSCs **`M2`** and **`M12`**, the order of the messages is not checked.

7.5 OptMsg

The example **`OptMsg.MPR`** (in the examples directory of the distribution) is an example that shows the semantics of an optional message. The implementation of **`doIt`** calls **`doItFinished`** after it's second call. Hence the first optional message will not be present, but the second instance will be present (see Figure 15)

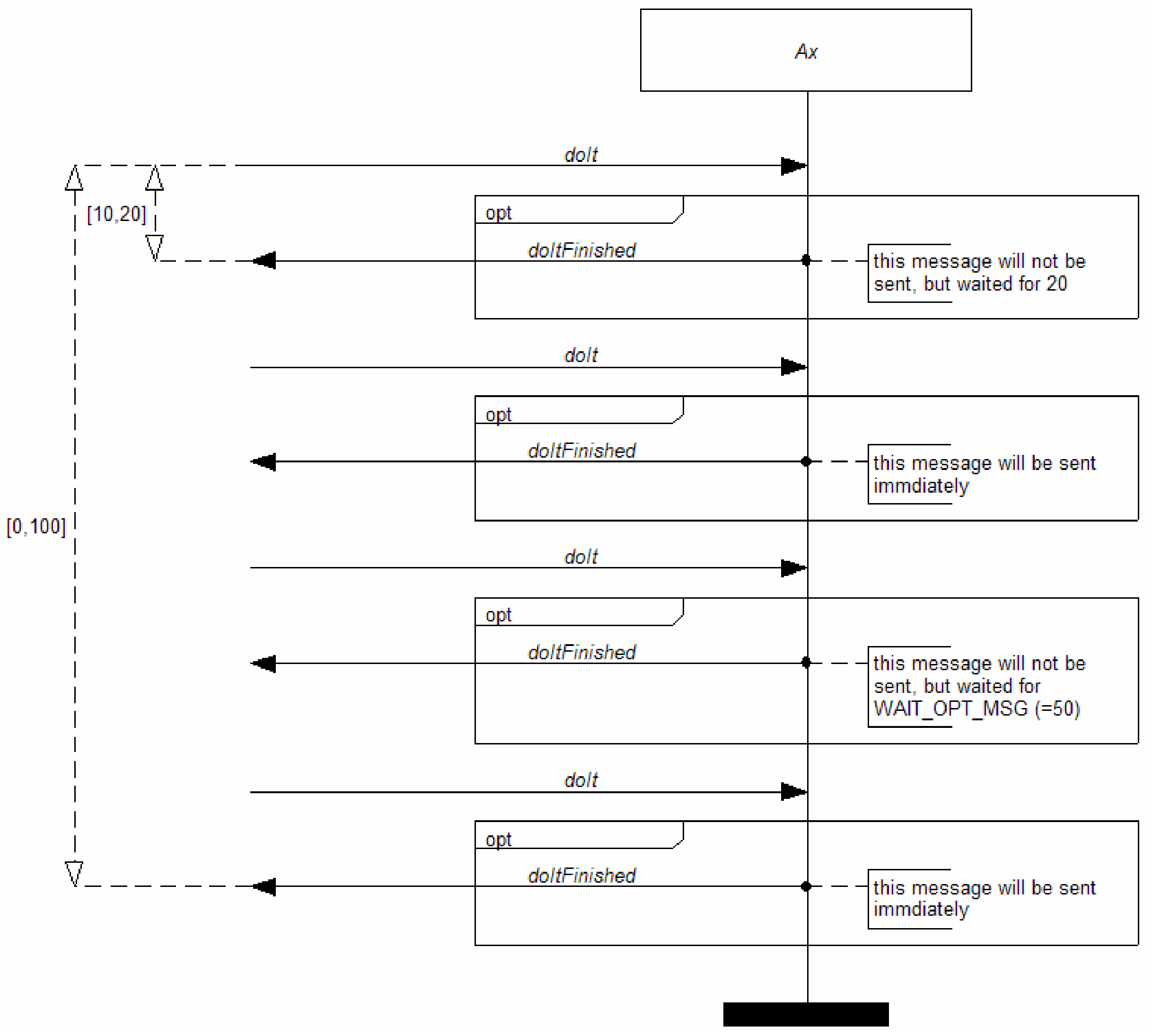

Ax
doIt
[10,20]
opt
doItFinished
this message will not be sent, but waited for 20
doIt
opt
doItFinished
this message will be sent immdiately
[0,100]
doIt
opt
doItFinished
this message will not be sent, but waited for WAIT_OPT_MSG (=50)
doIt
opt
doItFinished
this message will be sent immdiately

Figure 15: Example Optional Message

The result of the execution of this test shows not only the expected behaviour, but also how the time annotations are verified (see Figure 16).

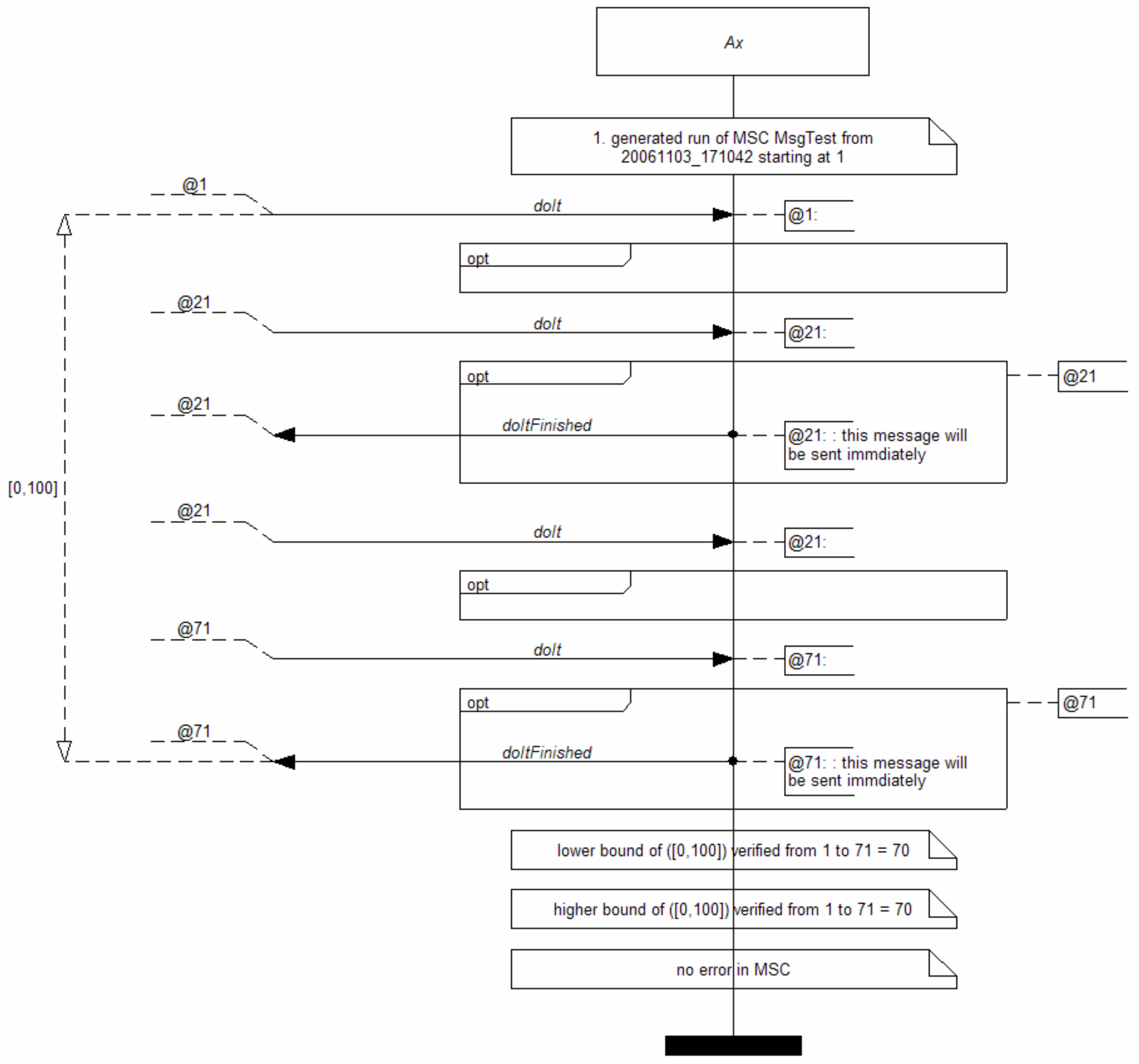

Ax
1. generated run of MSC MsgTest from 20061103_171042 starting at 1
@1
doIt
@1:
opt
@21
doIt
@21:
opt
@21
@21
doItFinished
@21: : this message will be sent immdiately
[0,100]
@21
doIt
@21:
opt
@71
doIt
@71:
opt
@71
@71
doItFinished
@71: : this message will be sent immdiately
lower bound of ([0,100]) verified from 1 to 71 = 70
higher bound of ([0,100]) verified from 1 to 71 = 70
no error in MSC

Figure 16: Result of Optional Message Test

8 Adaptations

The test code can be used in many different test environments without changing the MSC to C test code generator by adopting the wrapper code, for example to run the tests on an embedded target, or to test an SuT with a CAN or Flexray interface.

This section describes some typical adaptations like changes of the time concept or the tracing.

8.1 Formatter

The generated code is formatted, per default, by an external tool called GC GreatCode (http://sourceforge.net/projects/gcgreatcode/) in Version 1.140.

The formatter can be switched off using the option **-noformat** of **msc2c.bat**.

The formatter and the applied rules file can be found in the formatter directory: ***<your_install_path>*\Generator\Formatter**

The formatting rules can be adopted by changing **conventions.cfg** in the formatter directory. This file contains the command-line options of GC.

A list of all available options, can be obtained by executing **GC.exe -help** from a console in the formatter directory.

8.2 Types and Return Values

MSCs have only a rather untyped interface (the function catalogue) which describes only the names of the called and calling functions and their possible values (enumerated or as general number). MSC2C requires the correct types of such functions, for example to bind parameters into variables, or to generate the stubs. Therefore MSC2C assumes a simple and flexible type scheme: Every parameter **<p>** has the type **<p>_Type**. This can be defined in the function catalogue, e.g. by **#define p_Type** `uint8`. In order to reduce the amount of required type definitions for compiling the code, MSC2C generates defaults values for the types (in the file `TEST_types.h`): In the above example this is be:

```
    #ifndef p_Type
    #define p_Type DEFAULT_PARAM_TYPE
    #endif
```

The used default `DEFAULT_PARAM_TYPE` can also be defined in the function catalogue. The same principle hold for the return types of functions. The name of the return type for a message `<msg>` is `<msg>_Return_Type`.

The return values of stub functions (if present) are usually something like TRUE or E_OK. MSC2C cannot use different return values in messages that are incoming to the SuT (The reason for this is that the stub functions for the incoming messages returning the value are called asynchronously and do not have information in which MSC they are checked and which return value they should return in this MSC). Therefore MSC2C uses a default return value for each function, which is defined to the constant value `DEFAULT_RETURN_VALUE` in the file `msc2c_types.h`. If it is desired to have different return values for one stub within the test (for example to compute a check-sum, or to test an error case) the return value of the function has to be defined as a variable in the function catalog, e.g. by

```
#define msg_Return_Type int32
```

and

```
#define msg_Default_Return_Value my_msg_ReturnValue.
```

The variable `my_msg_ReturnValue` should be declared within the MSC and assigned a value in an action before the stub is modelled and the action which requires the return value from the stub is triggered, e.g. as described in Figure 17.

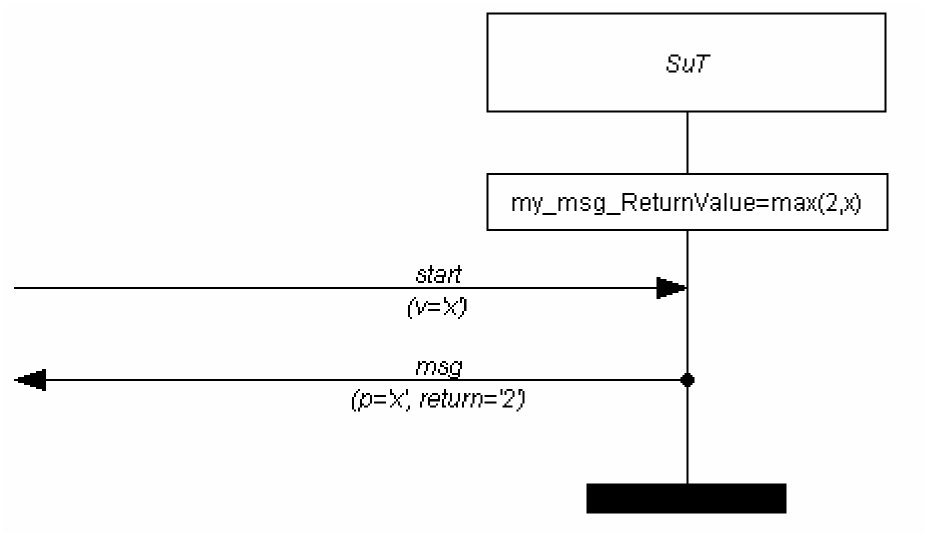


Figure 17: Changing Return Values of Stubs

Using complex structures for testing is also possible, however debugging and tracing requires to print the values of these types in a uniform way, e.g. using

```
printf("value of p = %d\n",p);
```

or

```
printf("value of p = %s\n",p);
```

This is achieved with the conversion macro **`TOVAL_p()`**, such that the values of **`p`** are printed by

```
printf("value of p = %d\n",TOVAL_p(p));
```

Like for **`p_Type`** MSC2C generates default implementations for the **`TOVAL`** macros (identity), which can be adapted by the user in the function catalogue. The selection whether **`"%d"`** or **`"%s"`** shall be used for the parameters is global for the test code. The default is **`"%d"`**. By defining **`#define VALUE_TYPE char*`** and undefining **`VALUE_TYPE_DECIMAL`** in the file **`msc2c_types.h`**.

For comparing elements of the types the EQUALS_Type functions are generated in the same way.

8.3 Specification Independent Code

This section provides the signatures for the adoptable functions. Details can be found in the code.

8.3.1 main

The file `main.c` contains the main routine for starting the test. As explained in Section 5.2.1, the test can also be called from other applications or frameworks.

8.3.2 msc2c_types

The file `msc2c_types.h` contains basic type definitions that can be changed, for example the type `DEFAUT_PARAM_TYPE` that is used as default for the parameter. Default types are used in functions, if no specific types are defined (see `TEST_types.h`).

8.3.3 msc2c_time

The file `msc2c_time.c` and `msc2c_time.c` contain the required functions for the timing. The default timing is a simple execution step counter, but other time concepts can be realized by changing this code.

```
/* type for getTime */
#define TIME_TYPE int

/**   * this action is executed in every step
  * (before the test is executed)
  */
void time_entry_action();

/** this action is executed in every step
  * (after the test is executed)
  */
void time_exit_action();

/**
  * this method returns the current time
  * (uint as used in the specification)
  */
TIME_TYPE getTime();

/**
  * this method actualizes the current time (if necessary)
  */
void RUN_TIMERS();
```

8.3.4 msc2c_debug

The file `msc2c_debug.c` and `msc2c_debug.c` contain the informations, which are relevant for debugging. Debugging differentiates between four groups of debugging functions:

- tracing functions that trace a model element (e.g. entering a state, receiving a message)
- debug info: addition information, e.g. .about values of timer, or priorities and coverage of MSCs
- error: information about occurred errors
- generation: information for the generation of MSC protocols

Every group of debugging information consists of sub-functions belonging to this group. Every group can be disabled / enabled separately. Furthermore for every group a start and an end point can be declared, for example to start the debugging information at a specified test time. All functions base on a small subset of macros, that can be easily changed for example if debugging information shall be written to a protocol file or a bus.

The following debugging functions are available:

```
/** this action is executed in every step (before the test is executed) */
 void debug_entry_action();
/** this action is executed in every step (after the test is executed) */
 void debug_exit_action();

/* type of detectable errors */
 typedef enum {
        OverallTestTimeout,
        ConditionViolated,
        UnexpectedMessage,
        MessageMissing,
        WrongMessageSequenceEarly,
        WrongMessageSequenceLate,
        WrongMessageParameterValue,
        WrongReturnParameterValue,
        WrongLoopRepetitionNumber,
    TimeAnnotationViolated,
    NoTimeoutOfTimer,
    WrongMSC2CConfiguration
   } ErrorType;

#define StepType uint16      /* limits the maximal test length to 65535 steps */
#define AktionType uint8     /* limits the maximal number elements in one MSC to 256 */

/* the error counter (is incremented by each error function)*/
 int iErrs;
/* the step counter (has to be incremented from the test-caller)*/
 StepType stCurStep;


/* general macros that are used from the generation functions */
 FILE* fGen; /* has to be initialized before using the generation, e.g. with an
fGen=fopen("file","w"); */
#define FG (fGen?fGen:stdout)
#define GEN_0(text) fprintf(FG,text);
#define GEN_1(text,a) fprintf(FG,text,a);
#define GEN_2(text,a,b) fprintf(FG,text,a,b);
#define GEN_3(text,a,b,c) fprintf(FG,text,a,b,c);
#define GEN_4(text,a,b,c,d) fprintf(FG,text,a,b,c,d);
#define GEN_5(text,a,b,c,d,e) fprintf(FG,text,a,b,c,d,e);
#define GEN_6(text,a,b,c,d,e,f) fprintf(FG,text,a,b,c,d,e,f);
#define GEN_7(text,a,b,c,d,e,f,g) fprintf(FG,text,a,b,c,d,e,f,g);
#define GEN_8(text,a,b,c,d,e,f,g,h) fprintf(FG,text,a,b,c,d,e,f,g,h);
#define GEN_9(text,a,b,c,d,e,f,g,h,i) fprintf(FG,text,a,b,c,d,e,f,g,h,i);
#define GEN_10(text,a,b,c,d,e,f,g,h,i,j) fprintf(FG,text,a,b,c,d,e,f,g,h,i,j);
```

```
/*
 * the following functions and variables can be used to
 * control the AMOUNT of debugging information
 * they may be called from the main routine
 */
 /* starts ALL at step s */
 void msc2c_set_ALL_start(StepType s);
/* stops ALL at step s */
 void msc2c_set_ALL_end(StepType s);
/* starts debuging at step s */
 void msc2c_set_debug_start(StepType s);
/* stops debuging at step s */
 void msc2c_set_debug_end(StepType s);
/* starts tracing at step s */
 void msc2c_set_trace_start(StepType s);
/* stops tracing at step s */
 void msc2c_set_trace_end(StepType s);
/* starts generation at step s */
 void msc2c_set_gen_start(StepType s);
/* stops generation at step s */
 void msc2c_set_gen_end(StepType s);
/* starts error at step s */
 void msc2c_set_error_start(StepType s);
/* stops error at step s */
 void msc2c_set_error_end(StepType s);

/* functions for controling features */
/**************************************/
/* control ALL debugging info */
 void msc2c_set_ALL(BOOL b);
 void msc2c_set_debug(BOOL b);
 void msc2c_set_trace(BOOL b);
 void msc2c_set_gen(BOOL b);
 void msc2c_set_error(BOOL b);

/* GENERIC */
 void msc2c_set_TTT_FFF(BOOL b);

/* debug: selected stubs  */
 void msc2c_set_debug_stub_all(BOOL b);
/* debug: all called stubs */
 void msc2c_set_debug_stub_selected(BOOL b);
/* debug: starting of timer */
 void msc2c_set_debug_timer_starting(BOOL b);
/* debug: reset of timer */
 void msc2c_set_debug_timer_reset(BOOL b);
/* debug: timeout timer  */
 void msc2c_set_debug_timer_timeout(BOOL b);
/* debug: coverage */
 void msc2c_set_debug_coverage(BOOL b);
 BOOL msc2c_get_debug_coverage();
/* debug: prio of MSC */
 void msc2c_set_debug_prio(BOOL b);
/* debug: priority of best MSC */
 void msc2c_set_debug_prio_best(BOOL b);
/* debug: idle message in HMSC-State */
 void msc2c_set_debug_idle_state(BOOL b);
/* debug: guards */
 void msc2c_set_debug_guards(BOOL b);
/* debug: guards with number */
 void msc2c_set_debug_guards_int(BOOL b);
/* debug: waiting */
 void msc2c_set_debug_waiting(BOOL b);
/* debug: calls */
 void msc2c_set_debug_calls(BOOL b);
/* debug: received calls from SuT (Stubs) */
 void msc2c_set_debug_received_calls(BOOL b);
/* debug: gates */
 void msc2c_set_debug_gates(BOOL b);


/* trace: hmsc */
 void msc2c_set_trace_hmsc(BOOL b);
/* trace: hmsc state */
 void msc2c_set_trace_hmsc_state(BOOL b);
/* prints trace: next hmsc-state */
```

```
 void msc2c_set_trace_hmsc_state_changed(BOOL b);
/* trace: msc */
 void msc2c_set_trace_msc(BOOL b);
/* trace: msc state */
 void msc2c_set_trace_msc_state(BOOL b);
/* trace: msc calls */
 void msc2c_set_trace_msc_call(BOOL b);
/* trace: msc finished */
 void msc2c_set_trace_msc_finished(BOOL b);
/* trace: msc with alternatives finished */
 void msc2c_set_trace_msc_alts_finished(BOOL b);
/* trace: msc calls in HMSC */
 void msc2c_set_trace_msc_call_hmsc(BOOL b);
/* trace: msc executed MSC in HMSC with coverage */
 void msc2c_set_trace_msc_called_hmsc_cover(BOOL b);
/* trace: selected msc calls in HMSC */
 void msc2c_set_trace_msc_call_hmsc_selected(BOOL b);
/* trace: msc pending (incomplete MSCs) */
 void msc2c_set_trace_msc_pending(BOOL b);

/* control genartion */
 void msc2c_set_gen(BOOL b);
/* gen: hmsc (complete HMSCs) */
 void msc2c_set_gen_hmsc(BOOL b);
/* gen: hmsc states */
 void msc2c_set_gen_hmsc_state(BOOL b);
/* gen: hmsc states */
 void msc2c_set_gen_hmsc_refs(BOOL b);
/* gen: msc (complete HMSCs) */
 void msc2c_set_gen_msc(BOOL b);
/* gen: msc calls */
 void msc2c_set_gen_msc_call(BOOL b);
/* gen: msc messages */
 void msc2c_set_gen_msc_message(BOOL b);
/* gen: msc actions */
 void msc2c_set_gen_msc_action(BOOL b);
/* gen: msc conditions */
 void msc2c_set_gen_msc_condition(BOOL b);
/* gen: msc timer */
 void msc2c_set_gen_msc_timer(BOOL b);
/* gen: msc time annotations */
void msc2c_set_gen_msc_timeann(BOOL b);
/* gen: msc hierarchic elements */
 void msc2c_set_gen_msc_hierarchic(BOOL b);
/* gen: msc hierarchic alternatives */
 void msc2c_set_gen_msc_hierarchic_alternative(BOOL b);
/* gen: msc hierarchic pars */
 void msc2c_set_gen_msc_hierarchic_par(BOOL b);
/* gen: msc hierarchic options */
 void msc2c_set_gen_msc_hierarchic_option(BOOL b);
/* gen: msc hierarchic loops */
 void msc2c_set_gen_msc_hierarchic_loop(BOOL b);


/* function that can be used in the code */
/*****************************************/
/* GENERIC */
 void msc2c_TTT_FFF();

/* debug: stub all */
 void msc2c_TTT_stub_all(const char*);
/* debugging of calls of stubs  */
void msc2c_debug_stub_all(const char* sStub);
/* debug: stub selected */
 void msc2c_debug_stub_selected(const char*);
/* debug: starting of timer */
 void msc2c_debug_timer_starting(char*,TIME_TYPE);
/* debug: reset of timer */
 void msc2c_debug_timer_reset(char*);
/* debug: timeout timer  */
 void msc2c_debug_msc_timer_timeout(const char* sMSC,const char* sTimer);
/* debug: priorities of MSCs */
 void msc2c_debug_prio(const char*,const char*,const char*,int);
/* debug: priority of best MSC */
 void msc2c_debug_prio_best(int);
/* debug: idle message in HMSC-State */
 void msc2c_debug_idle_state(const char*,const char*);
/* debug: guards */
```

```
 void msc2c_debug_guards(const char*,const char*);
/* debug: guards with int value */
 void msc2c_debug_guards_int(const char*,const char*,int);
/* debug: waiting  */
 void msc2c_debug_waiting(const char*,const char*,int);
/* debug: calls  */
 void msc2c_debug_calls(const char*,const char*);
/* debug: received calls from SuT (Stubs) */
 void msc2c_debug_received_calls(MSC_Type, MSG_TYPE);
/* debug: gates  */
 void msc2c_debug_gates(const char*,const char*);

/* prints trace: hmsc-state */
 void msc2c_trace_hmsc_state(const char* sHMSC,const char* sState,int iLevel);
/* prints trace: next hmsc-state */
 void msc2c_trace_hmsc_state_changed(const char* sHMSC,const char* sNewState);
/* prints trace: msc-state */
 void msc2c_trace_msc_state(const char* sMSC,int iMSCState);
/* prints tracing: msc calls */
 void msc2c_trace_msc_call(const char* sMSC);
/* prints tracing: msc finished */
 void msc2c_trace_msc_finished(MSC_Type iMSC,int iErrs);
/* prints tracing: msc with alternatives finished */
 void msc2c_trace_msc_alts_finished(const char* sMSC,int iAlts,int iErrs);
/* prints tracing: msc calls in HMSC */
 void msc2c_trace_msc_call_hmsc(const char* sHMSC,const char* sState,const char* sMSC);
/* prints tracing: executed msc in HMSC (with coverage) */
 void msc2c_trace_msc_called_hmsc_cover(const char* sHMSC,const char* sState,const char*
sMSC,int iCover);
/* prints tracing: selected msc calls in HMSC */
 void msc2c_trace_msc_call_hmsc_selected(const char* sHMSC,const char* sState,const char*
sMSC);
/* prints tracing: msc pending (incomplete MSCs) */
 void msc2c_trace_msc_pending(const char* sMSC,int iMSCState);

/* starts generation of an HMSC */
 void msc2c_gen_hmsc_start(int iHMSC);
/* ends generation of an HMSC */
 void msc2c_gen_hmsc_end(HMSC_Type iHMSC,int iErrors);
/* inserts an HMSCState */
 void msc2c_gen_hmsc_state(int iHMSC,int iState);
/* inserts a start of an MSC (MSC-Reference) */
 void msc2c_gen_hmsc_mscref(int iHMSC,int iMSC);
/* inserts a start of an MSC (MSC-Reference) with an alternative counter */
 void msc2c_gen_hmsc_mscref_alt(int iHMSC,int iMSC,int iAlts);
/* inserts an MSC Reference call (before the call) */
 void msc2c_gen_msc_call(int iMSCCalling,int iNode,int iMSCCalled);
/* inserts an MSC Reference call (after the call returned) */
 void msc2c_gen_msc_called(int iMSCCalling,int iNode,int iMSCCalled,int iErrors);
/* inserts an MSC Reference call (before the call) with an alternative counter */
 void msc2c_gen_msc_alt_call(int iMSCCalling,int iNode,int iMSCCalled,int iAlts);
/* inserts an MSC Reference call (after the call returned) with an alternative counter */
 void msc2c_gen_msc_alt_called(int iMSCCalling,int iNode,int iMSCCalled,int iAlts,int
iErrors);
/* inserts an action into the MSC */
 void msc2c_gen_msc_action(MSC_Type iMSC,int iAction);
/* inserts a condition into the MSC */
 void msc2c_gen_msc_condition(MSC_Type iMSC,int iNode,VALUE_TYPE vCond);
/* inserts a message with 0 parameters into the MSC */
 void msc2c_gen_msc_message0(MSG_TYPE iMSG,BOOL bDir,MSC_Type iMSC,int iNode);
/* inserts a message with 1 parameters into the MSC */
 void msc2c_gen_msc_message1(MSG_TYPE iMSG,BOOL bDir,MSC_Type iMSC,int iNode,VALUE_TYPE
iPar1);
/* inserts a message with 2 parameters into the MSC */
 void msc2c_gen_msc_message2(MSG_TYPE iMSG,BOOL bDir,MSC_Type iMSC,int iNode,VALUE_TYPE
iPar1,VALUE_TYPE iPar2);
/* inserts a message with 3 parameters into the MSC */
 void msc2c_gen_msc_message3(MSG_TYPE iMSG,BOOL bDir,MSC_Type iMSC,int iNode,VALUE_TYPE
iPar1,VALUE_TYPE iPar2,VALUE_TYPE iPar3);
/* inserts a message with 4 parameters into the MSC */
 void msc2c_gen_msc_message4(MSG_TYPE iMSG,BOOL bDir,MSC_Type iMSC,int iNode,VALUE_TYPE
iPar1,VALUE_TYPE iPar2,VALUE_TYPE iPar3,VALUE_TYPE iPar4);
/* inserts a message with 5 parameters into the MSC */
 void msc2c_gen_msc_message5(MSG_TYPE iMSG,BOOL bDir,MSC_Type iMSC,int iNode,VALUE_TYPE
iPar1,VALUE_TYPE iPar2,VALUE_TYPE iPar3,VALUE_TYPE iPar4,VALUE_TYPE iPar5);
/* inserts a timer start into the MSC */
 void msc2c_gen_msc_timer_start(MSC_Type iMSC,int iNode,int iTimer,int iValue);
/* inserts a timer timeout into the MSC */
```

```
 void msc2c_gen_msc_timer_timeout(MSC_Type iMSC,int iNode,int iTimer);
/* inserts a timer reset into the MSC */
 void msc2c_gen_msc_timer_reset(MSC_Type iMSC,int iNode,int iTimer);
/* inserts an exact time annotation (like [10]) between two nodes */
 void msc2c_gen_msc_timeann_start_exact(MSC_Type iMSC,int iNodeFrom,int iNodeTo,int iTimer,int
iValue);
/* inserts an interval time annotation (like [10,100]) between two nodes */
 void msc2c_gen_msc_timeann_start_inter(MSC_Type iMSC,int iNodeFrom,int iNodeTo,int iTimer,int
iValueMin,int iValueMax);
/* inserts an regular interval time annotation end */
 void msc2c_gen_msc_timeann_OK(MSC_Type iMSC,int iTimer,int iValue);
/* inserts an option begin into the MSC */
 void msc2c_gen_opt_begin(MSC_Type iMSC,int iNode);
/* inserts an option end into the MSC */
 void msc2c_gen_opt_end(MSC_Type iMSC,int iNode);
/* inserts an alternative begin into the MSC */
 void msc2c_gen_alt_begin(MSC_Type iMSC,int iNode);
/* inserts an alternative end into the MSC */
 void msc2c_gen_alt_end(MSC_Type iMSC,int iNode);
/* inserts an par begin into the MSC */
 void msc2c_gen_seq_begin(MSC_Type iMSC,int iNode);
/* inserts an par end into the MSC */
 void msc2c_gen_seq_end(MSC_Type iMSC,int iNode);
/* inserts and end of an MSC with the number of errors */
 void msc2c_gen_msc_finished(MSC_Type iMSC,int iErrs);

/* inserts a error into the MSC */
 void msc2c_gen_error_text(ErrorType error,int iErrorNumber,MSC_Type iMSC,int iNode,VALUE_TYPE
soll, VALUE_TYPE ist, const char* text);


 void dbg_error_text(ErrorType error,StepType step,MSC_Type msc,AktionType aktion, VALUE_TYPE
soll, VALUE_TYPE ist, const char* text);
```

9 Extensions

This chapter describes available extensions of the MSC2C testcode generator.

9.1 MSC Catalogue Generator

Some editors support the definition of message catalogues and allow the user to select instance and messages from those defined in the catalogue. Using catalogues increases the comfort of editing sequences. There are also standards like FIBEX for that purpose.

MSC2C supports a simple XML based catalogue for the definition of messages. In the MSC-Editor from ESG these catalogues can be used as "extension catalogues".

The generation of catalogues works using the command cataloggen and requires MPR files as input. For each MPR file a catalogue is generated that contains definitions for all instances used in the MSCs and all messages (including parameters). The parameters are enumerated by all values found in the document. If this is not desired the type (e.g. Number) should be inserted manually into the catalogue. The call is

```
cataloggen [MSC.mpr]*
```

The resulting catalogue is generated in a file MSC_cat.xml for each argument MSC.mpr.

9.2 Matlab Catalogue Generator

Similar to the generation of catalogues from MSCs it is also possible to generate catalogues from MATLAB/Simulink models.

The generated catalogue contains instances for all subsystems and atomic blocks that have interfaces (in or our ports). For every instance there exists the following messages:

- setAll_<Instance>: Message with all in ports as parameters
- set_<Port>: Message with one in port as parameter

- get_<Port>: Message with one out port as parameter

The get_<Port> and set_<Port> methods are generated for all ports in the model. This can lead to large number of instances and messages. For black box testing, where only the topmost interface is required, the option –frame is available. When this is specified only the outer elements are generated.

cataloggen [-frame] [**Model.mdl]***

The resulting catalogue is generated in a file Model_cat.xml for each argument Model.mpr.

9.3 Matlab Integration

The code generated from MSC2C can be used within Matlab/Simulink to test the models. In this case a MSC Block is inserted into the model and connected to the model under test (MuT).

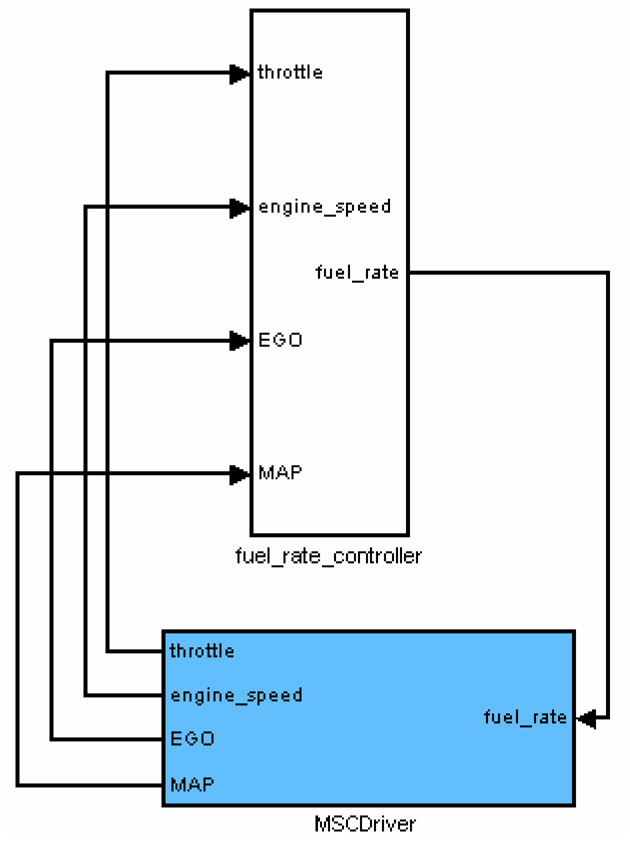
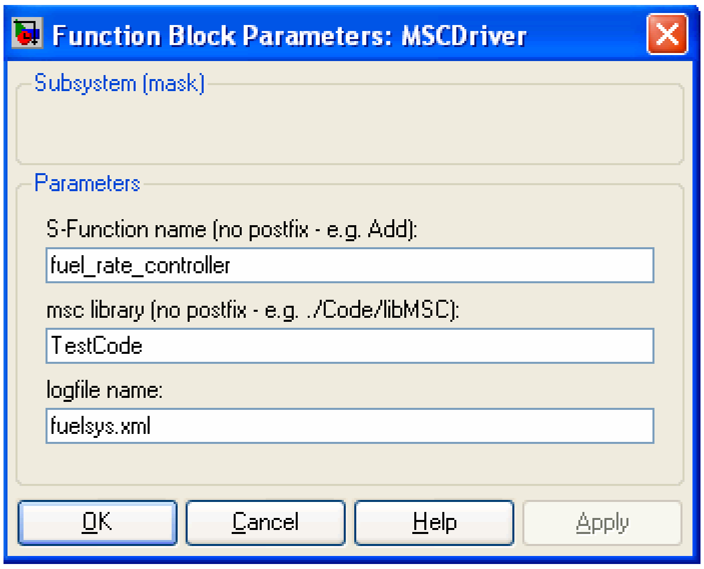


Figure 18: Integration of MSCs in Matlab/Simulink

For the development of the test the following steps are required:

1. Select (and isolate) the model under test

2. Generate a message catalogue for the MSC (see Section 9.2)
3. Edit the MSC using the message catalogue
4. Integrate the MSC into Matlab/Simulink and execute the test
    a. Generate code and s-function (using the MSC2C option –matlabtarget=)
    b. Compile the code and the s-function into a library (dll) with the command compileLib.bat (check pathes in this command!)
    c. Copy an MSCDriver from the mscdrivertemplate.mdl into the Model and adjust it's options to the generated and compiled code (see Figure 19)
    d. Connect MuT and MSCDriver by selecting the MuT and executing the m-script gen_stubs (from MSC2C/Matlab) in the Matlab target directory (where the compiled s-function is located)
    e. Execute the test by running the simulation


Function Block Parameters: MSCDriver

Subsystem (mask)

Parameters

S-Function name (no postfix - e.g. Add): fuel_rate_controller

msc library (no postfix - e.g. ./Code/libMSC): TestCode

logfile name: fuelsys.xml

Figure 19: Properteis of the MSC Driver

The MSC block has a parameter logfile, where a text file can be specified. If this parameter is present the test result will be documented into the selected file. This can be used for generating the MSC from the test.

An example for the Matlab integration is located in MSC2C/Examples/FuelSys.

9.4 MSC Transformations

MSC Transformations transform MSCs by replacing elements in a MSC according to specified replacement rules. I.e. a MSC transformation is a function t: MSC x Rule -> MSC, that takes a MSC document and replaces all elements according to the rules. The rules in general consist of a selection part and a transformation part. All elements in the MSC are compared with the selection part of a rule. If the selection part matches the rule fires by applying the transformation part to the matching element.

The implemented transformation tool uses MSCs for the specification of the transformation rules for messages. A MSC is interpreted as transformation rule in the form that the first element is used as selection criteria and if an element matches the first element it is replaced by all other elements in the MSC. The element matches, if it is of the same kind (Message or Condition) and if it has the same important attributes. The following attributes are checked during the selection:

- message:
    - name
    - direction
    - connected axes
    - parameter values
- condition:
    - text in the conditions (ignoring white spaces)

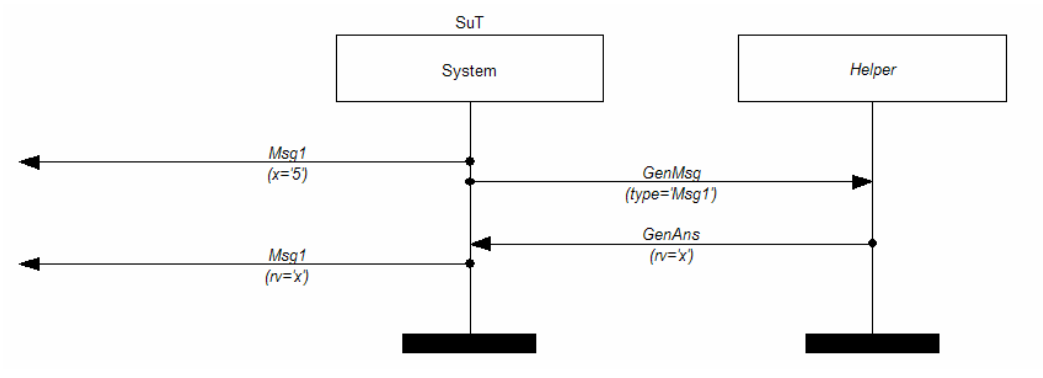


Figure 20: Transformation Rule

The MSC in Figure 20 describes a transformation rule for MSCs. The rule in the example will transform all messages Msg1 coming from the axis System which have the value 5 for the parameter x into the three messages GenMsg, GenAns and Msg1. While GenMsg and GenAns are new messages Msg1 is the matching message. The new message will be inserted as described (i.e. with one parameter), the matching message will be inserted with all parameters and the specified value.

Applying the rule in Figure 20 to the MSC described in Figure 21 results into the MSC in Figure 22, since only the second message matches.

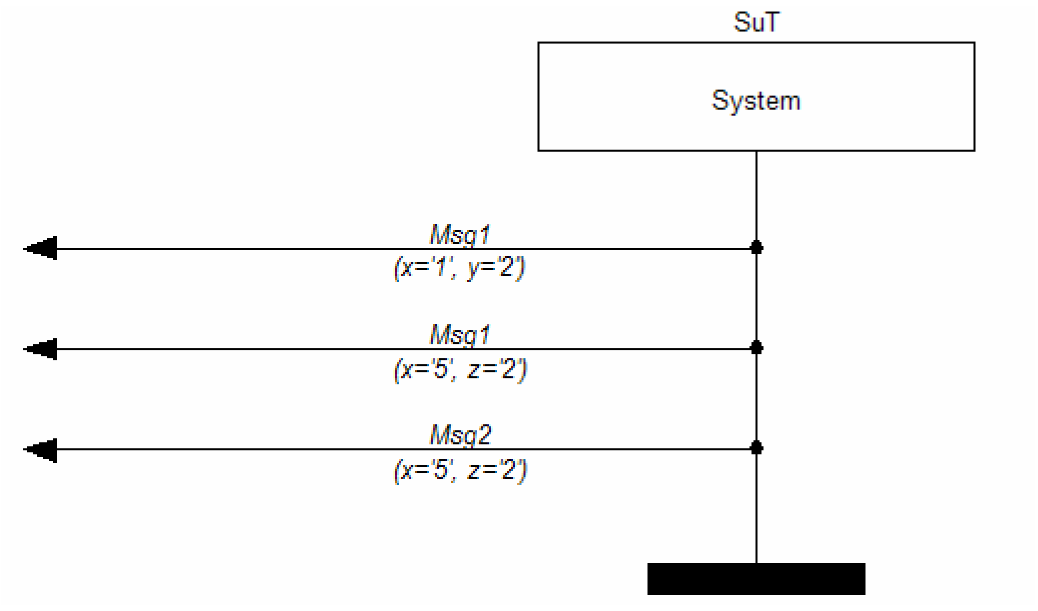


Figure 21: Original MSC

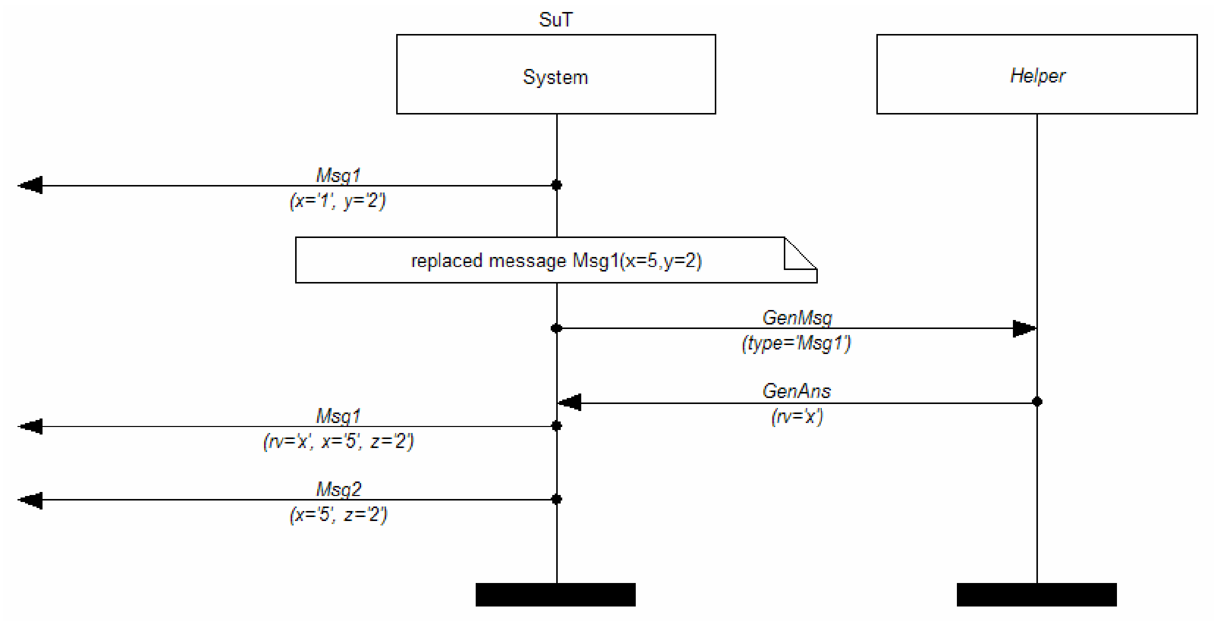


Figure 22: Transformed MSC

Note that all transformations in the transformation MSC are applied only once to the original MSCs. If several rules are specified the transformation order is the specification order in the MSC document for the rules. Only the first matching transformation rule is applied.

If the transformed elements have time annotations the starting time annotations attached to the last message of the new inserted elements. If the transformed element has ending time annotations they are attached to the first inserted message.

The MSCs can be transformed by calling the script

```
transform.bat MSCOrig.mpr Rules.mpr MSCNew.mpr [-elim]
```

If `MSCNew.mpr` is omitted `MSCOrig_trafo.mpr` will be created.

If the option `-elim` is specified all MSCs (and references to them) which contain no transformed message are eliminated from the transformation result.

10 Restrictions

Currently the checkgates option is not implemented and the named conditions are implemented as one variable for each condition, which is set but not reset. This shall be changed to one single variable that is used for all named conditions.

A. Appendix MPR Grammar

The MPR (MSC textual representation) grammar used for representing MSC is an implementation of the grammar specified by the ITU in Z120. The rules formatted in bold font are supported from the MSC parser, the other rules are formatted and overtaken from ITU, but cannot be used for the building of testable MSCs.

```
main ::= starcomment* mpr_file.
mpr_file ::= textual_msc_document message_sequence_chart*.
string ::= character_string
        |  name
        |  number.
index ::= name
       |  number.
end ::= comment? ';'.
comment ::= 'comment' character_string starcomment*.
text_definition ::= 'text' character_string starcomment starcomment* end.
textual_msc_document ::= document_head textual_defining_part
textual_utility_part.
document_head ::= 'mscdocument' instance_kind end using_clause*.
textual_defining_part ::= defining_language? defining_data?
defining_msc_reference*.
textual_utility_part ::= 'utilities' defining_msc_reference*.
defining_language ::= 'language' data_language_name end.
defining_data ::= 'data' character_string end.
defining_msc_reference ::= 'reference' msc_name.
using_clause ::= 'using' instance_kind end.
identifier ::= name.
message_sequence_chart ::= 'msc' msc_head msc_or_hmsc 'endmsc' end.
msc_or_hmsc ::= msc
             |  hmsc.
msc ::= msc_body.
msc_head ::= msc_name msc_parameter_decl? end msc_gate_interface.
msc_parameter_decl ::= '(' msc_parm_decl_list ')'.
instance_parameter_decl ::= 'inst' instance_parm_decl_list end.
instance_parm_decl_list ::= instance_parameter_name
co_instance_parm_decl_list?.
co_instance_parm_decl_list ::= ',' instance_parm_decl_list.
instance_parameter_name ::= instance_name.
message_parameter_decl ::= 'msg' message_parm_decl_list.
message_parm_decl_list ::= message_decl_list.
msc_parm_decl_list ::= msc_parm_decl_block end_msc_parm_decl_list?.
end_msc_parm_decl_list ::= end msc_parm_decl_list.
msc_parm_decl_block ::= data_parameter_decl
                     |  instance_parameter_decl
                     |  message_parameter_decl
                     |  timer_parameter_decl.
timer_parameter_decl ::= 'timer' timer_parm_decl_list.
timer_parm_decl_list ::= timer_decl_list.
instance_kind ::= identifier.
msc_gate_interface ::= msc_gate_def*.
msc_gate_def ::= 'gate' msg_gate end.
msg_gate ::= def_in_gate
```

```
          |  def_out_gate.
msc_body ::= msc_statement*.
msc_statement ::= text_definition
               |  event_definition.
event_definition ::= instance_name_list ':' event.
event ::= instance_event
       |  multi_instance_event_list.
instance_event_list ::= instance_event+.
instance_event ::= orderable_event
                |  non_orderable_event.
orderable_event ::= help0? help1 starcomment? end help2?.
help0 ::= 'label' event_name end.
help1 ::= message_event
       |  incomplete_message_event
       |  timer_statement
       |  action.
help2 ::= 'time' time_dest_list end.
time_dest_list ::= time_dest? time_interval co_time_dest_list?.
co_time_dest_list ::= ',' time_dest_list.
time_dest ::= event_name
           |  top_or_bottom reference_identification_or_label_name.
reference_identification_or_label_name ::= reference_identification
                                        |  label_name.
top_or_bottom ::= 'top'
               |  'bottom'.
non_orderable_event ::= start_coregion
                     |  end_coregion
                     |  instance_head_statement
                     |  instance_end_statement.
instance_name_list ::= instance_name comma_instance_name*
                    |  'all'.
comma_instance_name ::= ',' instance_name.
multi_instance_event_list ::= multi_instance_event+.
multi_instance_event ::= condition
                      |  msc_reference
                      |  inline_expr.
instance_head_statement ::= 'instance' instance_kind? starcomment? end.
instance_end_statement ::= 'endinstance' end.
message_event ::= message_output
               |  message_input.
message_output ::= 'out' msg_identification 'to' input_address.
message_input ::= 'in' msg_identification 'from' output_address.
incomplete_message_event ::= incomplete_message_output
                          |  incomplete_message_input.
incomplete_message_output ::= 'out' msg_identification 'to' 'lost'.
incomplete_message_input ::= 'in' msg_identification 'from' 'found'.
msg_identification ::= message_name co_message_instance_name?
parameter_helper?.
co_message_instance_name ::= ',' message_instance_name.
parameter_helper ::= '(' named_parameter_list ')'.
output_address ::= instance_name
                |  env_ref via_gate?.
env_ref ::= 'env'
         |  reference_identification.
via_gate ::= 'via' gate_name.
reference_identification ::= 'reference' msc_reference_identification.
input_address ::= instance_name
               |  env_ref via_gate?.
named_parameter_list ::= named_parameter_assignment
co_named_parameter_list?.
co_named_parameter_list ::= ',' named_parameter_list.
named_parameter_assignment ::= parameter_identification '=' parameter_value
parameter_value_binding?.
```

```
parameter_identification ::= parameter_name parameter_indices?.
parameter_indices ::= '[' index_name co_index_name? ']'.
co_index_name ::= ',' index_name.
parameter_value ::= string
                 |  wildcard
                 |  structured_parameter_value.
structured_parameter_value ::= '{' named_parameter_list '}'.
parameter_value_binding ::= right_bind_symbol pattern.
actual_out_gate ::= gate_name? 'out' msg_identification 'to' input_dest.
actual_in_gate ::= gate_name? 'in' msg_identification 'from' output_dest.
input_dest ::= 'lost'
            |  input_address.
output_dest ::= 'found'
             |  output_address.
def_in_gate ::= gate_name? 'out' msg_identification 'to' input_address.
def_out_gate ::= gate_name? 'in' msg_identification 'from' input_dest.
gate_identification ::= gate_name.
condition_identification ::= 'condition' condition_text.
condition_text ::= string
                |  'when' cond_expr
                |  'otherwise'.
cond_expr ::= condition_name_list
           |  '(' expression ')'.
condition_name_list ::= condition_name co_condition_name*.
co_condition_name ::= ',' condition_name.
condition ::= condition_identification end.
timer_statement ::= starttimer
                 |  stoptimer
                 |  timeout.
starttimer ::= kw_starttimer timer_name bounded_time? measurement?.
duration ::= '[' min_durationlimit? co_max_durationlimit? ']'.
co_max_durationlimit ::= ',' max_durationlimit.
durationlimit ::= expression_string
               |  'inf'.
stoptimer ::= kw_stoptimer timer_name co_timer_instance_name?.
co_timer_instance_name ::= ',' timer_instance_name.
timeout ::= 'timeout' timer_name co_timer_instance_name?.
action ::= 'action' action_statement.
action_statement ::= informal_action.
informal_action ::= character_string.
message_decl_list ::= message_decl end message_decl_list?.
message_decl ::= message_name_list.
message_name_list ::= message_name.
timer_decl_list ::= timer_decl end timer_decl_list?.
timer_decl ::= timer_name_list.
timer_name_list ::= timer_name.
variable_decl_list ::= variable_decl_item end variable_decl_list?.
variable_decl_item ::= variable_list ':' type_ref_string.
variable_list ::= variable_string.
data_parameter_decl ::= 'variables'? variable_decl_list.
actual_data_parameters ::= 'variables'? actual_data_parameter_list.
actual_data_parameter_list ::= parameter_name '=' expression_string
co_actual_data_parameter_list?.
co_actual_data_parameter_list ::= ',' actual_data_parameter_list.
right_bind_symbol ::= '=:'.
expression ::= expression_string.
pattern ::= variable_string
         |  wildcard.
wildcard ::= '_'.
time_point ::= '@'? time_expression.
measurement ::= rel_measurement
             |  abs_measurement.
rel_measurement ::= '&' time_pattern.
```

```
abs_measurement ::= '@' time_pattern.
time_interval ::= interval_label? bounded_time measurement?.
interval_label ::= 'int_boundary' interval_name.
singular_time ::= '[' time_point ']'
               |  measurement.
bounded_time ::= '@'? left_open_or_left_square_bracket time_point
comma_time_point? right_open_or_right_square_bracket.
left_open_or_left_square_bracket ::= '('
                                   |  '['.
right_open_or_right_square_bracket ::= ')'
                                     |  ']'.
comma_time_point ::= ',' time_point.
start_coregion ::= 'concurrent' end.
end_coregion ::= 'endconcurrent' end.
inline_expr ::= loop_expr
             |  opt_expr
             |  alt_expr
             |  seq_expr
             |  par_expr
             |  exc_expr.
loop_expr ::= 'loop' loop_boundary? 'begin' end msc_body 'loop_end' end.
opt_expr ::= 'opt_begin' end msc_body 'opt' 'end' end.
exc_expr ::= 'exc_begin' end msc_body 'exc' 'end' end.
alt_expr ::= 'alt_begin' end msc_body alts* 'alt' 'end' end.
alts ::= 'alt' end msc_body.
par_expr ::= 'par_begin' end msc_body pars* 'par' 'end' end.
pars ::= 'par' end msc_body.
seq_expr ::= 'seq_begin' end msc_body seqs* 'seq' 'end' end.
seqs ::= 'seq' end msc_body.
loop_boundary ::= '<' inf_natural co_inf_natural? '>'.
co_inf_natural ::= ',' inf_natural.
inf_natural ::= 'inf'
             |  expression.
msc_reference ::= 'reference' msc_reference_identification_colon?
msc_ref_expr end kw_time_time_interval_end? kw_top_time_dest_list_end?
kw_bottom_time_dest_list_end? reference_gate_interface.
msc_reference_identification_colon ::= msc_reference_identification ':'.
kw_time_time_interval_end ::= 'time' time_interval end.
kw_top_time_dest_list_end ::= 'top' time_dest_list end.
kw_bottom_time_dest_list_end ::= 'bottom' time_dest_list end.
msc_reference_identification ::= msc_reference_name.
msc_ref_expr ::= msc_name actual_parameters?.
actual_parameters ::= '(' actual_parameters_list ')'.
actual_parameters_list ::= actual_parameters_block
end_actual_parameters_list?.
end_actual_parameters_list ::= end actual_parameters_list.
actual_parameters_block ::= actual_data_parameters
                         |  actual_instance_parameters
                         |  actual_message_parameters
                         |  actual_timer_parameters.
actual_instance_parameters ::= 'inst' actual_instance_parm_list.
actual_instance_parm_list ::= actual_instance_parameter
co_actual_instance_parm_list?.
co_actual_instance_parm_list ::= ',' actual_instance_parm_list.
actual_instance_parameter ::= parameter_name '=' instance_name.
actual_message_parameters ::= 'msg' actual_message_list.
actual_message_list ::= parameter_name '=' message_name
co_actual_message_list?.
co_actual_message_list ::= ',' actual_message_list.
actual_timer_parameters ::= 'timer' actual_timer_list.
actual_timer_list ::= parameter_name '=' timer_name co_actual_timer_list?.
co_actual_timer_list ::= ',' actual_timer_list.
reference_gate_interface ::= end_gate_ref_gate*.
```

```
end_gate_ref_gate ::= 'gate' ref_gate end.
ref_gate ::= actual_out_gate
          |  actual_in_gate.
hmsc ::= 'expr' msc_expression.
msc_expression ::= start node_expression*.
start ::= label_name_list? end.
node_expression ::= label_name ':' node_helper end.
node_helper ::= node_helper2 'seq' '(' label_name_list? ')'
              |  'end'.
node_helper2 ::= timable_node
               |  node.
label_name_list ::= label_name alt_label_name*.
alt_label_name ::= 'alt' label_name.
node ::= condition_identification
       |  'connect'.
par_expression ::= 'expr' msc_expression 'endexpr' par_helper*.
par_helper ::= 'par' 'expr' msc_expression 'endexpr'.
timable_node ::= '(' msc_ref_expr ')' time_helper?
               |  par_expression time_helper?.
time_helper ::= 'time' time_interval end.
msc_name ::= string.
instance_name ::= string.
event_name ::= name.
message_name ::= string.
message_instance_name ::= string.
gate_name ::= name.
timer_name ::= name.
timer_instance_name ::= name.
interval_name ::= name.
inline_expr_name ::= name.
condition_name ::= string.
label_name ::= name.
message_sequence_chart_name ::= string.
msc_reference_name ::= string.
data_language_name ::= name.
parameter_name ::= string.
par_name ::= name.
index_name ::= index.
sdl_document_identifier ::= identifier.
variable_identifier ::= identifier.
expression_string ::= string.
type_ref_string ::= string.
variable_string ::= string.
wildcard_string ::= string.
data_definition_string ::= string.
max_durationlimit ::= durationlimit.
min_durationlimit ::= durationlimit.
time_expression ::= expression.
time_pattern ::= pattern.
open_par ::= character_string.
close_par ::= character_string.

// important lexical helpers:
///////////////////////////

name ::= ( letter | underline ) name_element*.
kw_stoptimer ::= 'stoptimer' | 'reset'.
starcomment ::= ( '//' not_eol* eol ) | ( '/*' not_star* '*'+ (
not_star_slash not_star* '*'+ )* '/' ).
number ::= decimal_digit+.
kw_starttimer ::= 'starttimer' | 'set'.
character_string ::= ''' ( all_but_quote | quote quote )+ '''.
```