	Model-based Testing for Matlab/Simulink	Technical Notice TN MTest-2001-6

MWATCH Installation and Users Guide

Markus Lepper
— ÜBB/TU Berlin —

mw0-2.tut —DRAFT— 19.06.2002

markuslepper.eu

markuslepper.eu

Contents

1 MWATCH Principles **5**

2 Installation and Configuration in a MATLAB/simulink Environment **5**
2.1 Installation 5
2.2 Changing the directory/file locations 6

3 Using MWATCH **7**
3.1 MWATCH Blocks 7
3.2 Manual Installation of an MWATCH block 8
3.3 Resolving of file names – Windows NT version 10
3.4 The MWATCH API for Program Controlled Watchdog Installation . . 10
3.5 The MWATCH Console for Controlling Multiple Predicates from One File for One Subsystem 10

4 Structure of an MWATCH Specification File **13**
4.1 File Sections representing different MWATCH Predicates 13
4.2 Comment lines 14

5 The MWATCH Language of Temporal Predicates **14**
5.1 Accessing simulation data via ⟨*TestPoint*⟩s 14
5.2 Instantaneous Predicates 15
5.3 Syntax and Semantics of the Temporal Combinators 15
5.4 Auxiliary Language Constructs 17
5.4.1 Defining testdata 17
5.5 The MWATCH Macro Facility 18
5.5.1 Lexical Scopes and Shadowing 18
5.5.2 Resolving of macros 19

6 Issues of Practical Operation **23**
6.1 Principles of the MATLAB/simulink/MWATCH Interface 23
6.2 Timing Considerations and Treatment of a „Final Verdict“ 23
6.3 Persistency 24
6.4 Error Diagnosis 24

7 Little Example and Tutorial **25**
7.1 Example Text 25
7.2 Explanations 29
7.2.1 29
7.2.2 30
7.2.3 31
7.2.4 31

A Embedding into MTest **32**

List of Figures

1 Operation Specification for Installing MATCH 6
2 Operation Specification for Running a Simulation with an MATCH-Block Operated through the GUI 8
3 NICHT MEHR GANZ AKTUELL !!! Layout of the MATCH blocks mask. 9
4 Syntax of the MATCH top level file structure. 13
5 The grammatical circles of the MATCH language 20
6 Schematic Grammar of the MATCH Predicate Language. 21
7 Schematic Grammar of the MATCH Macro Facility. 22

List of Tables

1 The MATCH API for programmed control of blocks 11
2 The MATCH API related to Specification Files 12
3 List of supported MATLAB/simulink library functions 16

---

simulink is a trademark of „The Mathworks,Inc.“, Natick, MA, USA
MATLAB is a trademark of „The Mathworks,Inc.“, Natick, MA, USA

---

markusleppler.eu

1 MWATCH Principles

MWATCH is a tool for the evaluation of test data traces against a given temporal specification.

A systems behavior – given as a sequence of tuples of values, each tuple represents a system state sample and is tagged with a time stamp, – is checked against a term which denotates a set of legal traces.

This checking is implemented dia-chronuosly, so that any violation of the specification will be detected as soon as possible with respect to the consumed data, and out-of-time as well as realtime application are feasible.

While the MWATCH *algorithm* is of course totally independent from the kind of system to check, its actual *implementation* is done as a „block set“ in the MATLAB/simulink environment. An MWATCH block can be inserted into a simulink model to watch the behavior of the model during a simulation run and create the verdict in simulated real time.

Currently our implementation supports MATLAB version `5.3.1(.29215a)`, i.e. MATLAB `R11.1`, together with „simulink 3“.

The following sections describe the installation and operation of this implementation, while section 5 describes the MWATCH language and semantics and section 7 explains a small example operating on the model „`sf_car`“, which is contained in the MATLAB/simulink distribution.

2 Installation and Configuration in a MATLAB/simulink Environment

2.1 Installation

All you have to do for installing is unzip the distributed `.zip`-archive and add some directories to your MATLAB-search-path. Additionally you can adjust some parameters in the „`Configuration`“ block in the MWATCH blockset.

The file structure of the MWATCH`.zip` archive is [1] . . .

- directory `visible` :
  - the MWATCH blockset library (`MWatchLib.mdl`),
  - diverse MatLab function files (`mw`<*xxxx*>`.m`),
  - the executive „S-Function“ implementation (`mwdoeval4.dll`),
  - the MWATCH-compiler (`mw02.exe`),
- directory `demo` :
  - some demo files (<*xy*>`.mw`, <*xy*>`.mat` and <*xy*>`.mdl`)
- directory `doc` :
  - this file.

[1] The names of the files will probably change slightly in future releases, but except the name of the library you really do not need to know them `:-)`

markuslepper.eu

Figure 1 Operation Specification for Installing MWATCH

```
   unzip the distributed .zip file to directory <mwdir>
-> (   add the directory  "<xmwdir>/visible" to your Matlab search path
   | add the directory  "<xmwdir>/tmp" to your Matlab search path
   | ? add the directory  "<xmwdir>/demo" to your Matlab search path
   )
-> Call "MWatchSetup"  from the Matlab command prompt.
-> The library opens
-> Do "unlock library" in the libraries menu
-> Press "return" at the Matlab command prompt
-> ? Adjust the default settings in the configuration block
```

The sequence for installing MWATCH is as follows :

1. Unzip the MWATCH-archive into an arbitrarily named directory. (The path of this directory will be referred to as <*mwatchdir*> in the sequel.)
2. Add the directory <*mwatchdir*>`/visible` to your MATLAB search path[2].
3. Add the directory <*mwatchdir*>`/tmp` to your MATLAB search path.
4. If you want to run the *demos*, you must additionally add the directory <*mwatchdir*>`/demo` to your MATLAB search path.
5. Call `MWatchSetup` from the MATLAB command prompt.
   `MWatchSetup` will set the paths and locations used for executing MWATCH automatically depending on the location of <*mwatchdir*>.
   The values will be stored in the parameters of the `configuration` block in the `MWatchLib` library. Since there is no (documented) possibiliy to unlock an open library by programmed commands, you have to do the unlocking manually when prompted to do so.

After this installation, and if the ⟨*mwatchDir*⟩`/demo` directory is included in the MATLAB path, the functionality of MWATCH can easily be tested by issuing the command (**preliminary :**)

```
mwDemo (4)
```

from the MATLAB command line. A demo model shall open, an MWATCH-block will be inserted and launched, and when running the simulation the block shall turn red or green.

2.2 Changing the directory/file locations

The parameter values of the block `MWatchLib/configuration` can be adjusted according to your needs at any time[3].

[2] This can be done either by starting `pathedit` from the MATLAB prompt, or by adding a `path` command to your MATLAB startup script, – please refer to the MATLAB/simulink documentation for details.

[3] Actually there are only two parameters, but maybe there are more to come . . .

markusslepper.eu

Especially if you change the position of the `tmp` directory (since this is the only one which must be *writable* and therefore should not reside in the read-only MWATCH-domain) you must put the new position onto the MATLAB search path.

3 Using MWATCH

To run a simulink-session with MWATCH-verification active, you must

1. create a file containing the specification to be checked against.
2. install one ore more MWATCH blocks into the system to be watched.
3. run the simulation as usual.

For the first step any text editor can be used, since an MWATCH file is an ordinary ASCII text file[4] containing a specification in the MWATCH language. The structure of such a file is described in section 4 and the constructs and meanings of the MWATCH language are described in detail in section 5.

The last step is performed as usual.

For the second step there are three possibilities :

1. Installing an MWATCH block manually by „drag'n'drop“ and entering the the file system location of the text file into a field of the „mask“ of this block.
2. Install an MWATCH block with a given specification file by calling the MWATCH API.
3. Installing an MWATCH Console which allows easy selection of different predicates from one single file, as long as they are related to the same (sub-)system, and generates an XML coded test protocol.

3.1 MWATCH Blocks

The central means for linking a simulink model to the MWATCH algorithm is the installation of an MWATCH block into this system.

Each MWATCH block has two (2) outports called `pass` and `fail`. These are of type boolean and will turn to `true` as soon as a pass or fail of the specification is detected. An MWATCH block is a „masked subsystem“, into which the instantaneous predicates (will be explained later in 5) will be compiled as a simulink network, and which contains an „S-function“ block realizing the evaluation algorithm as a `.DLL`.

The system immediately containing a certain MWATCH block is the „Sub-System Under Test“ from this MWATCH blocks „point of view“. It will be referred to by „SSUT“ in the following. All naming of signals and ports occuring in a specification linked to a certain MWATCH-block is always resolved *relatively* to this SSUT.

An MWATCH block does not have any inputs. All signals which are to be watched are fed by „`From`“ and „`Goto`“ blocks into the subsystem. Those blocks will be inserted automatically into the SSUT and will be removed as soon as the MWATCH block is deleted.

[4]The lexer currently in use does not support larger character sets, i.e. „Latin-1“ or „unicode“ is not supported.

Figure 2 Operation Specification for Running a Simulation with an MWATCH-Block Operated through the GUI

```
   provide a SIMULINK model for being tested
-> (   write one or more  MWatch file(s)  with specs for this model
     | ( open topsystem under test -> open subsystem to test )
     | open MWatch library )
-> drag'n drop an Mwatch block into SSUT
-> double click on MWatch block for getting the mask
-> enter filename or path (maybe with section name) into textfield
-> * (click onto "edit" -> *(edit the file -> save the file) )
-> ( click onto "compile and load"
      -> system: does compilation and linkage
     | ? set checkbox "hold on fail"
     | ? set checkbox "hold on pass"
   )
-> ? close mask using "OK"-button
-> run simulation
```

Every signal and value contained in the toplevel of the SSUT can be watched, as well as those in any arbitrarily deep subsystem of the SSUT, as long as all *library links* leading to this subsystem *are „broken"*.

3.2 Manual Installation of an MWATCH block

Simply open the MWATCH library as usual (e.b. by issueing „`open mwatchlib`" from the MATLAB command prompt) and „drag'n'drop" the MWATCH Block into the system to check. You can install more than one MWATCH block into the same system, thus checking it against different specifications simultaniously.

For each block you have to identify the specification to check against by giving the position of the specification file in the file system, maybe together with the name of a section. Further you can influence the behavior of the MWATCH block by additional input fields in the block's mask.

Double-clicking on the MWATCH block opens the parameter mask as shown in figure 3.

The path of the MWATCH source file to check against has to be entered into the topmost text field. The way how file pathes are resolved is operating system dependent and explained in section 3.3.

The check buttons below are *abused* as command buttons, because such are not foreseen in a simulink mask, e.g. clicking on these boxes will make a check mark appear and be deleted again immediately, after the command is executed.

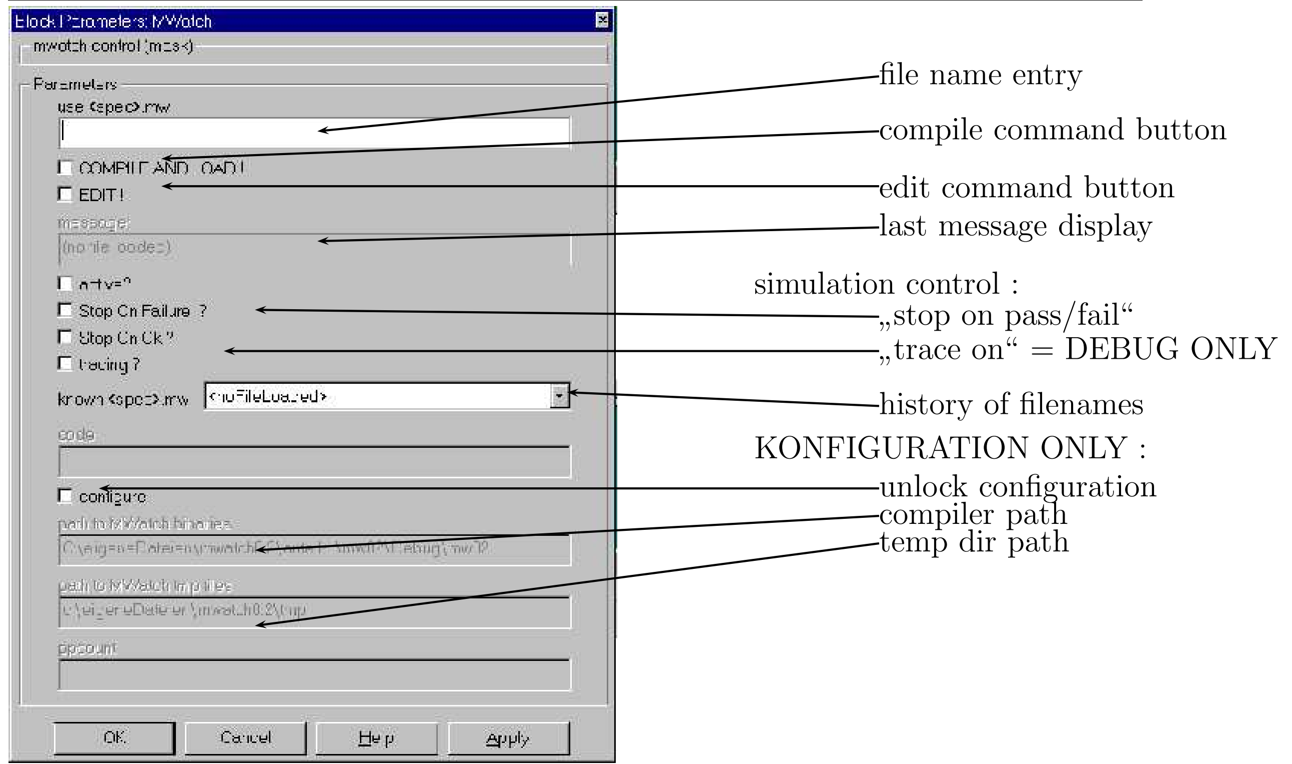
The „Edit"-button will launch the MatLab Editor with the given file.

The „Compile&Load"-button will start the MWATCH compiler. After successful compilation the MWATCH box will be launched with the specification, thus appearing in *yellow* color in the systems display.

Before the compilation is started, any existing launching of the MWATCH box

markusslepper.eu

Figure 3 NICHT MEHR GANZ AKTUELL !!! Layout of the MWATCH blocks mask.



will be cleared, and the MWATCH box will appear in *white* color.

If there are any errors in compiling or launching, – e.g. syntax errors, invalid block names, etc. – the error message appears in the „message“ window and is echoed the MatLab command window.

preliminary : A modal dialog could be opened too !!??

The MWATCH block will stay *white* (=not launched) in these cases.

If the file contains *sections* (see below 4.1), the file name has to be followed by the name of the section to be used, preceeded by a hashmark sign (`#`).

All files which have been compiled and launched successfully are stored in the *file name history*. Selecting a line in the „file name history selection field“ just copies the text into the file name entry field, so the procedure of compiling, launching, editing etc. must be performed in the same way as if the file name had been entered character by character[5].

To disactivate an MWATCH-box just open the selection box of the file name history and select the pseudo file „`<no file selected>`“

The check buttons „stop on failure“ and „stop on success“ enable MWATCHs possibility for aborting a simulation run[6].

When running a simulation any MWATCH box in the system under test will

[5]**preliminary :** Since there is no documented way of forcing such a selection box to synchronize with its underlying data model, a file name successfully installed will not appear visibly in the selection list until you close and open the mask, *but is already entered* into the data. So do not wonder if you miss the name of the last file and get an „off by one“ error when selecting a file name from the list `;-(`

[6]**preliminary :** The check button „trace“ is for internal debugging purposes only.

turn *green* as soon as the fact of passing the specification is established, and turns *red* as soon as the system fails the specification.

The verdict (PASS or FAIL) and the verdict time is displayed in the message line of the ///ATCH-blocks mask, and (**preliminary :**) in two dedicated output fields in the mask.

3.3 Resolving of file names – Windows NT version

All files for ///ATCH (either specifications given to ///ATCH-blocks or data files, see below) are identified either by paths or by pure file names.

Three flavours are supported :

- File names or paths starting with „./“ are interpreted relatively to the „current directory“ of the „current drive“. This specification is *not recommended*.
- File names or paths starting with „⟨*Driveletter*⟩:/“ are interpreted as absolute pathnames.
- All other file names or paths must *not* start with „/“ and are searched for in all diretories listed in the path variable internal to MATLAB.

3.4 The ///ATCH API for Program Controlled Watchdog Installation

There is a small API providing MATLAB functions for installation, control and de-installation of ///ATCH blocks from the MatLab command prompt or any function written in the MATLAB-language („M-File“). The functions of this API are listed in table 1 on page 11. The API and the GUI are[7] fully compatible, i.e. it is possible to install a ///ATCH-block using the API, modify its parameters via GUI, delete it via API etc.

Please note that a subsystem must be given by it full path name relative to its top level system. All functionalities and parameter interpretations are the same as said above with the manual installation, and file names are resolved as described in section 3.3.

The return values of the API functions are always starting with a pair of integer and text string. The integer value 0(zero) denotes success, in which case the text string is of no interest. All integer values $\neq 0$ indicate an *error*, and the text will be set accordingly to an error description.

In addition the API contains some functions not related to a distinct block, which are explained *in situ* in table 2 on page 12.

3.5 The ///ATCH Console for Controlling Multiple Predicates from One File for One Subsystem

When a given ///ATCH file contains different predicates for a single (sub)system, an „///ATCH Console“ can be installed for switching on and off the evaluation of the different predicates.

This is realized by automatic insertion and deletion of ///ATCH blocks.

[7] ...should be ;-)...

markusklepper.eu

Function		Signature / Description
`mwInstall`	⇒	(⟨*SubsystemName*⟩, ⟨*MWatchFileAndSect*⟩, ⟨*blockName*⟩) [⟨*ErrorCode*⟩, ⟨*ErrorText*⟩, ⟨*MwBlockPath*⟩] Installs a new MWATCH-block with given name for given spec.file. In case of success returns the full path of the newly created MWATCH-block. If an empty ⟨*blockName*⟩ (a MATLAB array of length=0) is given, a fresh name is supplied automatically.
`mwRelink`	⇒	(⟨*MwBlockPath*⟩, ⟨*MWatchFileName*⟩) [⟨*ErrorCode*⟩, ⟨*ErrorText*⟩] Changes the task of the block to watch the given specification. If the compiling of the specification results to an error, the blocks old linkage is kept unchanged.
`mwStopsim`	⇒	(⟨*MwBlockPath*⟩, ⟨*Integer*⟩, ⟨*Integer*⟩) [⟨*ErrorCode*⟩, ⟨*ErrorText*⟩] Sets the influence of the MWATCH-block to the run of the simulation : giving a value of „0“/„1“ as the first (second) parameter disables/enables the interruption of the simulation in case of fail (pass).
`mwDeinstall`	⇒	(⟨*MwBlockPath*⟩) [⟨*ErrorCode*⟩, ⟨*ErrorText*⟩] Removes the MWATCH-block.
`mwGetverdict`	⇒	(⟨*MwBlockPath*⟩) [⟨*ErrorCode*⟩, ⟨*ErrorText*⟩, ⟨*verdict*⟩, ⟨*verdictTime*⟩] Returns the verdict and time of decision of the given MWATCH-block as delivered in the last simulation run. The verdict is encoded as *text string* : '`FAIL`' '`PASS`' '`inconc`' The verdict time is encoded as MATLAB float value. **preliminary :** The special value `99999.99` is used to indicate a verdict recognized *past the end* of a simulation run.

Table 1: The MWATCH API for programmed control of blocks

markuslepper.eu

`mwMaketemplate`	(⟨*SubsystemPrefix*⟩, ⟨*Block*⟩, ⟨*FilePath*⟩)
⇒	[⟨*ErrorCode*⟩, ⟨*ErrorText*⟩] Creates a new file with the given path and name, containing macro definitions for all testable ⟨*ValuePoints*⟩ in the subsystem identified by the concatenation [⟨*SubsystemPrefix*⟩"/"⟨*BlockName*⟩]. All Ports and Output connectors of all blocks contained in this system are enumerated. The definition of their identifiers is *relative* to the given ⟨*SubsystemPrefix*⟩, therefore appropriate for installing an MWATCH block at exactly this level.
`mwGetsections`	(⟨*FilePath*⟩)
⇒	[⟨*ErrorCode*⟩, ⟨*ErrorText*⟩,⟨*Sectionlist*⟩] Scans the given file and delivers a „cell array" of all section names contained in this file, ordered by their first appearance.
`mwNewconsole`	(⟨*FilePath*⟩, ⟨*SubsystemPath*⟩)
⇒	[⟨*ErrorCode*⟩, ⟨*ErrorText*⟩, ⟨*FigureNumber*⟩] Creates a new „MWATCH Predicate Selection Console" for the subsystem with the given path and the file with the given name. The file has to contain predicates relatively defined to this subsystem. A new window (MATLAB „figure" object) is created and displayed. The figure number returned identifies this window when calling figure-related MATLAB functions.

Table 2: The MWATCH API related to Specification Files

An MWATCH Console can only be created using the API (function „`mwNewconsole`") and is realized as a MATLAB „`figure`" object, i.e. a seperate top-level window controlled by MATLAB.

An MWATCH Console (see figure FEHLT) consists of a panel in which each line corresponds to a „section" in the file. Left to the name of the section a checkbox can be found, to the right is a text area for displaying the status of the evaluation.

Below this panel there are some command buttons which provide the following functionality:

- Button „`Edit`" :
  Calls the editor for the file.
- Button „`Rescan File`" :
  Rescans the file and updates the list of sections.
  This should be used if, after editing the file, its contents have changed regarding the set of section names, e.g. a new section has been entered. The file is re-scanned and the list of sections adjusted accordingly.
- Button „`Do Install`" :
  Installs an MWATCH block for each section on which a *check mark* is set, and deinstalls all others.
  This should be used before starting a simulation run. For all predicates which could be compiled without error a new MWATCH block is inserted and the textfield containing the section name turns *yellow*. White name fields indicate non-installed predicates.
- Button „`Update Verdicts`" :

markusslepper.eu

Copies the verdicts from the MWATCH blocks to the text area to the right of the sections name and additionally signals the verdict by color.
This should be used after completion of a simulation run. The verdict is printed right to the name textfield, and the color of the textfield is changed according to the verdict.

- Button „`Write Protocol`“ :
  A new protocol file is created or an existing protocol file expanded by an XML encoding of the verdicts of the last simulation run.
  The name of the protocol file is derived from the name of the attribute file in an operating-system-dependent manner.
  In the case of windows, where no multiple dots are allowed, we derive
  <*mwatchdir*>`/demo/sf_car.mw`
  ...
  <*mwatchdir*>`/demo/sf_car_mw_results.xml`

4 Structure of an MWATCH Specification File

4.1 File Sections representing different MWATCH Predicates

Figure 4 Syntax of the MWATCH top level file structure.

⟨*mwatchFile*⟩	==	⟨*mwatchText*⟩
	|	$\langle mwatchText \rangle^?$ $\langle fileSection \rangle^+$
⟨*fileSection*⟩	==	`#section` ⟨*sectionName*⟩ ("," ⟨*sectionName*⟩)* (":")? "\n" ⟨*mwatchText*⟩ "\n"
⟨*mwatchText*⟩	==	⟨*any text not containing* "`#section`" ⟩
⟨*sectionName*⟩	==	⟨*any alphanumeric identifier, underscore* "_" *may be used*⟩

Above the „language level“ an MWATCH file may contain different „sections“, see the grammar in figure 4. These are used to represent multiple predicates in a single file, maybe sharing common definitions.

Each section begins with the keyword „`#section`“, followed by a comma seperated sequence of section names. A section name is an arbitrary chosen identifier consisting of digits, numbers or the underscore „_“; it may begin with or consist entirely of numeric characters.

Each file section extends up to the next `#section` keyword or to the end of file, which ever comes first. Any predicate text selected by a section name is made up from the concatenation of the file text preceding the first "`#section`" keyword (if present), and all file sections containing this section name in the section line.

please note : that the newline character "\n" is significant, i.e. a section line cannot expand over more than one text line, and no ⟨*mwatchText*⟩ may appear in the same line as a "`#section`" keyword.

A specification file may or may not contain `#section` keywords.

If not, no section must be selected whenever launching an MWATCH box, and all the contents of the file make up the specification.

If the file does contain a `section` keyword, a valid section identifier *must* be given together with the file name whenever evaluating this file by MWATCH.

4.2 Comment lines

Comments are supported only on a one-line base : All characters between the character "`&`" or the character pair "`//`" up to the end of a line are treated as comments.

5 The MATCH Language of Temporal Predicates

The MATCH language allows to formulate temporal predicates based on a „Trace Semantics" :

Each term of an MATCH formula describes a „segment of time", and the combinators of the language are used to combine these segments.

Basic building block is the ⟨*InstPred*⟩ construct, which stands for any „timeless", instantaneous boolean predicate. It represents all segments of the trace in which the predicate is fulfilled in every time instant belonging to the intervall.

A given trace *fulfills* the MATCH-specification iff *at least* one segmentation of the test trace can be found, in which each segment fulfills one ⟨*InstPred*⟩, and the order of the adjacent segments fulfills the semantics of the combinators applied to the corresponding instantaneous predicates.

The instantaneous predicates are *implemented* by creating an invisible network of simulink-blocks inside the MATCH-block; thus their evaluation is delegated to MATLAB/simulink, and the exact semantics can be derived from their documentation.

The temporal combinators are evaluated by a `.DLL`, and their semantics are described in more detail in the following.

5.1 Accessing simulation data via ⟨*TestPoint*⟩s

All predicates in an MATCH specification are built on observations of values, changing in time during the simulation run. In the grammar of our specification language these values are identified by the nonterminal ⟨*TestPoint*⟩. The instantaneous predicates mentioned aboce are made up watching these ⟨*TestPoint*⟩s.

As ⟨*testPoint*⟩ may serve :

- The name of a simulink block, immediately followed ba a slash (/) and an Output Port *number*.
  If the block has exactly *one* output port, this number and the seperating slash may be omitted.
  This numeric adressing can used to access output signals of instances of the *built-in* primitive simulink blocks, the ports of which are not accessible by name.
- The name of a simulink subsystem block, immediately followed ba a slash (/) and the *name* of an Output Port.
- The name of a „signal" or „line" somewhere in the model.

Signals and blocks can be deeply buried in subsystems of the SSUT, in which case they are named by giving the complete path (relative to the SSUT) seperated by slashes. (Beware not to put blanks around these slashes, as otherwise they would be recognized as „divide" operators.)

markusklepper.eu

Any identifier starting with an alphabetic character and consisting only of alphas, decimal digits and the special characters `#`, `_`, `/`, `.`, `'` and `-` can be written down directly.

All other port or signal identifieres, e.g. containing exclamation marks or blanks, have to be included in double quotes ("`"`"). With this notation nearly *all* character combinations can be used. Only when using this notation there is a rather primitive expansion mechanism to generate *newline* characters, since all character sequences „`\n`" are reposed by a newline char[8].

In case that the name of one subsystem in a path contains special characters, blanks or newlines, the *whole path* has to be included in double quotes. Beware that no pruning of quoted strings is implemented, and (**please note :**) that some of the built-in blocks of simulink contain blanks which are *not visible* because they appear at the end or beginning of a line.

5.2 Instantaneous Predicates

The boolean expressions representing ⟨*InstPred*⟩s are either

1. directly used ⟨*TestPoint*⟩s, if the correspondig simulink port or signal is of *type boolean*, or
2. ⟨*comparison*⟩s of arithmetic expressions built from numeric constants, ⟨*testPoint*⟩s and timed test data from MATLAB data files, i.e. ⟨*xy*⟩`.mat` files), or
3. boolean expressions formed by the operators "`||`", "`<=>`", "`=>`", "`&&`", and "∼" (increasing binding power) applied to the formers.

Numeric constants can be integer or floating points. Floating points can be given in scientific notation. Sequences of digits not containing a decimal seperator are recognized as integer values and cannot contain an exponential part. **please note :** There is no „unary minus" operator, but one minus sign can be entered as first character of a numeric constant. So there is a difference between „`5 - 3`", which is correct, and „`5 -3`" which is not.

Arithmetic expressions are built either with the usual MATLAB-operators, or by calling the library functions listed in table 3 on page 16.

5.3 Syntax and Semantics of the Temporal Combinators

On top of these instantaneous expressions (= ⟨*instPred*⟩) the language elements describing sets of traces are built :

- A simple ⟨*instPred*⟩ statement matches all those (sub-)traces for which in *each time instance* the given instantaneous predicate does hold.
- The predicate `ANY` matches *all* traces[9].

[8] **preliminary :** It is not possible to use the character sequence „`\n`" *verbatim* in an identifier, because there is no sophisticated escaping mechanism.

[9] Remark for the reader familiar with the discussion of the semantics: Because of the „continuity hypothesis" the `ANY` *seems* to match the empty trace, but indeed it matches only „infinitively short" subtraces of arbitrary content.

markuslepper.eu

Function	Syntax / Description
`diff`	"(" ⟨*arithExpr*⟩ ")" Derivation of given signal
`irgd`	"(" ⟨*arithExpr*⟩ ")" *Discrete* integration of given signal (Initial condition set to `0.0`.)
`abs`	"(" ⟨*arithExpr*⟩ ")" Absolute value.
`min` `max`	"(" ⟨*arithExpr*⟩("," ⟨*arithExpr*⟩)$^+$ ")" Calculate min/max of list of signals.
`sin` `cos tan asin acos atan atan2 sinh cosh`	"(" ⟨*arithExpr*⟩ ")" Trigonometric functions and inverses
`delay`	"(" ⟨*arithExpr*⟩"," ⟨*arithExpr*⟩ ")" Delay the first signal dynamically; the duration of the delay is determined by the second signal (**ToBeDone:** maxdelay/samplecount is set to default value, – add parameters !?)
`shold`	"(" ⟨*arithExpr*⟩"," ⟨*boolExpr*⟩ ")" Sample-and-hold the former signal; re-sampling is triggered by the latter
`memory`	"(" (⟨*arithExpr*⟩ | ⟨*boolExpr*⟩) ")" Memorize the signal from the *last simulation step*. **Notice:** The simulink documentation forbids to use this block together with certain solvers (`ode15s` and `ode113`)
`scope`	"(" ⟨*integerConst*⟩"," ⟨*integerConst*⟩ "," (⟨*arithExpr*⟩ | ⟨*boolExpr*⟩) ")" Send the given signal to one channel of an implicitly created multi-channel scope device. The first ⟨*integerConst*⟩ determines the „pane“ of the scope, the second the „channel“ where to send the signal.
`wspi`	"(" ⟨*Ident*⟩ ","⟨*integerConst*⟩")" Creates a simulink „`fromWorkspace`“ block. ⟨*Ident*⟩ is used immediately for the mask parameter „`variableName`“, and so its interpretation is exclusively defined by simulink. No checks on the validity of this ident is performed by MATCH! The ⟨*integerConst*⟩ gives the channel number of this newly created device, the value of which is used as the value of the expression.
`file`	"(" ⟨*Ident*⟩ ","⟨*integerConst*⟩")" Creates a simulink „`fromFile`“ block. ⟨*Ident*⟩ is used to identify the file, which has to be of "`.mat`" type. The rules for resolving file pathes are the same as above (3.3). The ⟨*integerConst*⟩ gives the channel number of this newly created device, the value of which is used as the value of the expression.

Table 3: List of supported MATLAB/simulink library functions

markusslepper.eu

From this simple semantic definition it is clear that `ANY` can only be useful together with some combinators, e.g. as part of a „chopped“ sequence or constrained by `MIN` or `MAX`.

- Any expression built with the „chop“ operator ”;” holds for those traces which can (at some *arbitrarily chosen* point) be split into two segments, the first satisfying the first formula, the second the second.
- Expressions (either simple or built with combinators) prefixed by `MIN` or `MAX` match only those traces with given minimal or maximal length. The length has to be given as real or integer constant.
  please note : Expressions are not yet permitted here `;-(`.
- Two or more expressions (either simple or built with combinators themselves) combined with `CASES`... `AND`... `AND`... denotate the set of traces wich fulfill all given predicates.
  Expressions combined with `CASES`... `OR`... `OR`... denotate the set of traces wich fulfill at least one of the given predicates.
- The `REP` prefix denotates sets of runs, each of which can be seperated into arbitrary many sub-runs, each fulfilling the prefixed formula. This does *not* include the empty trace. The `REP` prefix resembles the `+` construct known from regular expressions.
  The `REP` prefix can only be used *on top* of sequences built with the chop operator ”;”, since the repetition of an instantateous predicate is idempotent with the predicate itself, – a `MAX` constraint even simply vanishes when being repeated.
- The `OPT` prefix can only be used *beneath* a sequence of segments built with the chop operator ”;”. It denotes either the traces fulfilling the prefixed formula, or the empty trace.
  please note : that even if given no `MIN` constraint, the mere appearance of an instantaneous predicate requires that this predicate is *valid* in some arbitrarily small intervall in time. Each ⟨*instPred*⟩ can be seen as implicitly contained in a „`MIN` ϵ ...“ statement, with $\epsilon > 0$.
  The `OPT` operator permits this ϵ to be really equal to zero (0.0).
- The combinations `OPT REP` or `REP OPT` denote repetitions *including* the empty trace, thus realizing the `*` construct known from regular expressions.
- The `REPN` prefix denotates those runs which can be divided into exactly n segments, each of which fulfills the prefixed formula, where n is given as integer constant[10]
  please note : Expressions are not (yet) permitted here `;-(`.

5.4 Auxiliary Language Constructs

5.4.1 Defining testdata

The statements of form

`SET` ⟨*testPoint*⟩ ”`=`” (⟨*arithExpr*⟩ | ⟨*boolExpr*⟩) ”`;`”

where ⟨*testPoint*⟩ has to be the path of a signal or an *output* port, *changes* the

[10] Since this construct is expanded in the compiler, and the primitive sequentialization interface does not support *sharing* at the moment, the integer value should be not too large.

markuslepper.eu

SSUT by replacing the values sequence produced originally by those produced by the given ⟨*arithExpr*⟩ or ⟨*boolExpr*⟩.

preliminary : The implementation of our model extractor, needed to find the coordinates of the original signal line, is rather straightforward and **almost unacceptable slow**.

preliminary : ATTENTION The information for **undoing** any `SET` command is not stored persistently. Please *do not save* a model containing a `from` tag resulting from executing a `SET` expression.

5.5 The MATCH Macro Facility

As a means for *abstraction* the MATCH-language includes a versatile macro mechanism.

please note : that, as mentioned above, when evaluating an MATCH-file with a given section name, the prefix of the file *preceeding* the first „`section`. . . " line and *all* file segments the headline of which include the given section name will be concatenated to form the predicate text for evaluation.

So it is possible to include all macros and definitions used by different predicates in such shared sections.

The macro mechanism is characterized by these properties:

- It does not support recursion.
- It needs definition-before-use.
- It supports locally defined macros.
- It operates on syntactic elements, not on character level.

As you may have noticed, the grammar in figure 6 does contain the top level syntax for *defining* macros by „`LET` ⟨*ident*⟩ "`=`" ⟨*defBody*⟩" etc., but neither the syntax (1) for instantiating („calling") them, nor (2) for the ⟨*defBody*⟩ itself.

This is because (1) nearly every production of this grammar can be substituted by an instantiation of an appropriate macro, and (2) a macro definition body is just a normal MATCH-expression, where these same productions can be replaced by the appearance of a macro parameter. It follows from this, that (3) also the arguments of any macro call can be made of arbitrary derivations from these same productions, as long as the macro call *expands to a syntactically valid* construct.

The grammar of macro definition and usage can be depicted as in figure 7.

5.5.1 Lexical Scopes and Shadowing

Since our macro mechanism permits nested macro definitions, the rules for resolving lexical references have to consider the hierarchy of definitions. We follow the usual approach that „later" definitions override the „earlier" ones, i.e. each lexical entity is looked for by ascending the hierarchie starting at the place of application.

The precedence is as follows :

- When in a macro definition the highest precedence comes to the *parameter names*.
- Thereafter it is looked for a *macro definition* of the next higher nesting level, *ignoring* the name of the active macro itself.

markusslepper.eu

- This is repeated until we left the scope of the highest macro.
- Then the identifier is looked up in the table of built-in functions (see table 3).
- *Only* those identifiers not matched with parameter names, macro names or built-in library functions are treated as port or signal names.
  But this should case no problems, as *in praxi* a port or outlet identifier causes at least one slash ("/"), and slashes can easily be avoided[11] in the name of macros and macro parameters.

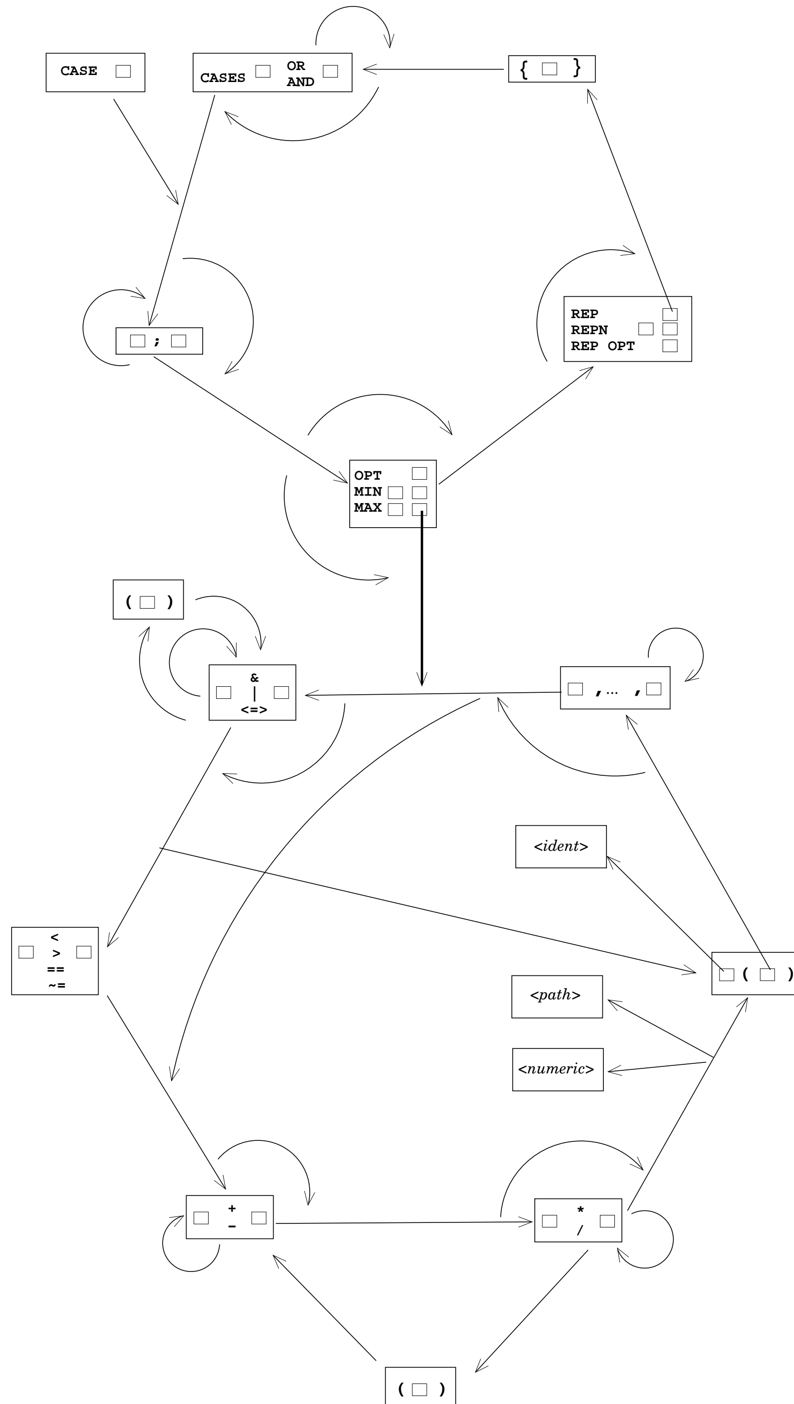
5.5.2 Resolving of macros

◯ <FIXME><FEHLT>MORE TO COME

[11] should be PROHIBITED!?

markusslepper.eu

Figure 5 The grammatical circles of the MATCH language



markusslepper.eu

Figure 6 Schematic Grammar of the MWATCH Predicate Language.

```
⟨mwatchTest⟩     ==  ⟨testCaseDef⟩* ⟨localDef⟩* ⟨mwatchSpec⟩

⟨testCaseDef⟩    ==  SET ⟨testPoint⟩ "=" ⟨expr⟩ ";"
⟨localDef⟩       ==  LET ⟨ident⟩      "=" ⟨defBody⟩
                  |  LET ⟨ident⟩ "("⟨ident⟩ ( "," ⟨ident⟩)* ")"
                                      "="  ⟨defBody⟩

⟨mwatchSpec⟩     ==  ⟨combination⟩ EOF
                  |  CASE⟨sequence⟩ EOF
⟨combination⟩    ==  CASES⟨sequence⟩(OR⟨sequence⟩)+
                  |  CASES⟨sequence⟩(AND⟨sequence⟩)+
⟨sequence⟩       ==  ⟨qualified⟩(";"⟨sequence⟩)*
⟨qualified⟩      ==  MIN⟨numConst⟩⟨primOrSeq⟩
                  |  MAX⟨numConst⟩⟨primOrSeq⟩
                  |  OPT⟨primOrSeq⟩
⟨primOrSeq⟩      ==  ⟨instPred⟩
                  |  ⟨compound⟩
                  |  REP⟨compound⟩
                  |  REPN⟨integerConst⟩⟨compound⟩
                  |  REP OPT⟨compound⟩
⟨compound⟩       ==  "{"⟨combination⟩"}"
                  |  "{"⟨sequence⟩"}"

⟨instPred⟩       ==  ⟨boolExpr⟩

⟨boolExpr⟩       ==  ⟨boolExpr⟩ ("&&"|"||"|"="|"=>"|"<=>") ⟨boolExpr⟩
                  |  "~"⟨boolExpr⟩
                  |  "("⟨boolExpr⟩")"
                  |  ⟨testPoint⟩   // if simulink-type of outport is „boolean“
                  |  ⟨comparison⟩

⟨comparison⟩     ==  ⟨arithExpr⟩ ("<"|"<="|"="|">"|">=") ⟨arithExpr⟩

⟨arithExpr⟩      ==  ⟨arithExpr⟩ ("+"|"-"|"*"|"/") ⟨arithExpr⟩
                  |  "("⟨arithExpr⟩")"
                  |  ⟨numConst⟩
                  |  ⟨testPoint⟩   // if simulink-type of outport is „simple numeric“
                  |  ⟨fileData⟩
                  |  ⟨simulinkMatlabFunction⟩ "(" ⟨arglist⟩ ")"
                     // these functions are listed in table 3
⟨argList⟩        ==  ⟨expr⟩(","⟨expr⟩)*
⟨expr⟩           ==  ⟨boolExpr⟩ | ⟨arithExpr⟩

⟨numConst⟩       ==  ⟨integerConst⟩|⟨floatConst⟩

⟨testPoint⟩      ==  (⟨subsysName⟩"/")*⟨blockName⟩("/"⟨portNumber⟩)?
                  |  (⟨subsysName⟩"/")*⟨blockName⟩"/"⟨portName⟩
                  |  (⟨subsysName⟩"/")*⟨signalName⟩
```

Figure 7 Schematic Grammar of the MATCH Macro Facility.

⟨*macroDefinition*⟩	==	`LET` ⟨*ident*⟩ "`=`" ⟨*defBody*⟩ "`;`"
	|	`LET` ⟨*ident*⟩ "`(`"⟨*ident*⟩ ("`,`" ⟨*ident*⟩)* "`)`"
		"`=`" ⟨*defBody*⟩ "`;`"
⟨*defBody*⟩	==	⟨*timedExpression'*⟩
	|	⟨*timedStatement'*⟩
	|	⟨*arithExpr'*⟩
	|	⟨*boolExpr'*⟩
⟨*timedExpression'*⟩	==	... (like ⟨*timedExpression*⟩, but all referred nonterminals primed) ...
	|	⟨*parameterIdent*⟩
⟨*boolExpr'*⟩	==	... (like ⟨*boolExpression*⟩, but all referred nonterminals primed) ...
	|	⟨*parameterIdent*⟩
⟨*arithExpr'*⟩	==	... (like ⟨*arithExpr*⟩, but all referred nonterminals primed) ...
	|	⟨*parameterIdent*⟩
⟨*numConst'*⟩	==	... (like ⟨*numConst*⟩) ...
	|	⟨*parameterIdent*⟩
⟨*testPoint'*⟩	==	... (like ⟨*testPoint*⟩ , but all referred nonterminals primed) ...
	|	⟨*parameterIdent*⟩
⟨*macroCall'*⟩	==	... (like ⟨*macroCall*⟩, but all referred nonterminals primed) ...
	|	⟨*parameterIdent*⟩
⟨*timedExpression*⟩	==	... (like figure 6 above) ...
	|	⟨*macroCall*⟩
⟨*boolExpr*⟩	==	... (like figure 6 above) ...
	|	⟨*macroCall*⟩
⟨*arithExpr*⟩	==	... (like figure 6 above) ...
	|	⟨*macroCall*⟩
⟨*numConst*⟩	==	... (like figure 6 above) ...
	|	⟨*macroCall*⟩
⟨*testPoint*⟩	==	... (like figure 6 above) ...
	|	⟨*macroCall*⟩
⟨*macroCall*⟩	==	⟨*ident*⟩
	|	⟨*ident*⟩ "`(`" ⟨*macroArg*⟩ ("`,`" ⟨*macroArg*⟩) ∗ "`)`"
⟨*macroArg*⟩	==	⟨*timedExpression*⟩
	|	⟨*boolExpr*⟩
	|	⟨*arithExpr*⟩
	|	⟨*testPoint*⟩

markusslepper.eu

6 Issues of Practical Operation

6.1 Principles of the MATLAB/simulink/MWATCH Interface

Whenever the user requests the launching of an MWATCH-block with a specification, the compiler is called and generates a MATLAB script.

This script is generated in a known, dedicated temporal directory, thus passed to the MATLAB interfacing software.

Then this script is executed and (1) generates a „masked subsystem“ under the MWATCH-block. This subsystem realizes the calculation of the „instantanuos predicates“, i.e. the arithmetic and logic expressions underlying the temporal predicate. Data file imports are also realized „unvisibly“ in this subsystem.

The script (2) sets some parameters of the mask of the MWATCH-block, thereby transporting the count of input ports, and the coding of the „time grammar“ MWATCH is going to parse.

Finally (3) the script really does *modify* the SSUT by adding „`Goto`“-blocks to the signals which are used by the specification. If the feature of test data generation is used, it furthermore adds „`From`“-blocks into the SSUT, thereby *disconnecting* the original data source.

Whenever an MWATCH-block is deleted or relaunched, all these manipulations are reverted.

When a simulation is started, (4) the `.DLL` realizing the MWATCH algorithm reads some values from the block mask, decodes the serialized time grammar and builts the data structures needed to evaluate the predicate given by the launched specification.

6.2 Timing Considerations and Treatment of a „Final Verdict“

In simulink it seems that the count how often the model function „`mdlOutputs()`“ ic called can vary with the kind of solver used to execute the simulation. Therefor we implemented an „off-by-one“ discipline: the MWATCH S-Function is called as soon as the time stamp of this calling increases, and evaluates the values of the *last* time of calling. So each verdict will be calculated one step behind the time when it became valid (but of course semantically with the correct time stamp value).

This normally does not cause any problem, since these verdicts are a kind of meta-information, which can be read (e.g. via API) after the simulation has ended. Speaking strinctly some kinds of verdicts can *only* be calculated together with the meta-information, *that* the simulation run is finished, e.g. „`ANY ;` *p* `EOF`“ needs this information. So what you really read after the end of a simulation run with the function `getVerdicts()` is the *final verdict*, – a verdict which is calculated under the premise, that the simulation in finished.

So *if* you want to use the verdicts, which are given to the Aboth outports of the MWATCH block as boolean values, on the object level, e.g. connecting those outputs to other blocks in some testbed, these final verdicts have to be calculated in already in the very last step of the simulation run.

This in accomplished by activating the button „`include final verdict`“ in the MWATCH block's mask.

please note : that all mask values are copied from the state of the mask of the block in the library MWatchLib whenever a new block is installed, so check and adjust this „prototype“ mask.

6.3 Persistency

Any simulink-model containing MWATCH-blocks can be stored to and read from disk without any problems, even if those are „launched“ with a specification. The MATLAB-file containing the compiled specification is only needed temporarily, while executing the „compile & load“ command.

What should *not* be done is editing the „masked subsystem“ of the MWATCH-block or changing the value of the „*gotoTag*“-field of an MWATCH-generated goto or from-block, since this can disturb the mechanism of undoing the modifications.

preliminary : ATTENTION: The information for undoing goto blocks inserted due to a SET statement is not (yet) persistent, see 5.4.1.

6.4 Error Diagnosis

In the sequence of launching an MWATCH-block with a certain specification there are diverse stages at which different classes of errors can be recognized :

- . . . during compilation :
  The compiler detects (of course) all syntactical errors. It detects some semantic errors, e.g. minimal timings with a duration larger than a maximal timing on the same expression, timing constraints less than or equal to zero etc.
- . . . during „loading“ of the compiled code :
  Since the compiler has *no knowledge* of the model for which the specification is compiled, all „port not found“ errors are detected when executing the compiler generated MATLAB-file. The same is true for „port number required“ or „library link not broken“ etc.
  But since from the users point of view both phases seem to be a single activity, this distinction causes no problems.
- . . . when starting the simulation :
  Each simulink-model can contain ports and signals of boolean and of floating point type. The somehow peculiar design of simulink detects type mismatches of connections *not before* the simulation is *started* !
  This is rather annoying and leads to the fact that the semantic error of class „type mismatch“ between signals and input ports in an MWATCH-specification can only signalled by simulink, and *not before starting the simulation*.
  please note : Furthermore there is *no (documented) way* of determining the count of channels stored in a (time-stamped) .MAT-file! So the error of using a too large channel number in a file() function call will also be detected *not earlier* than the start of the simulation.

markusslepper.eu

7 Little Example and Tutorial

7.1 Example Text

Entering at the MATLAB command line the command

mwDemo (0)

opens a demonstration model from the MATLAB/simulink/stateflow distribution, and installs an MWATCH-console for applying the predicate file

<*mwatchdir*>/demo/sf_car.mw

Here is the predicate file's text :

Source Example 1

```
1... // file : "<MWatchDir>\demo\sf_car.mw"
2... // MWatch specification example for simulink(c)/stateflow(c) model "sf_car"
3... // author : The MWatch Team (DC-FT3/TUB-UeBB)
4...
5... LET vspeed= "vehicle\nspeed" ;
6... LET brake = "brake\nschedule" ;
7... LET gas   = "throttle\nschedule" ;
8... LET gear  = shift_logic/gear ;
9... LET vsp'  = diff (vspeed) ;
10..
11.. LET S (a,b,c) = scope (a,b,c) ;
12.. LET S11 (a) = S(1,1,a) ;
13.. LET S12 (a) = S(1,2,a) ;
14.. LET S13 (a) = S(1,3,a) ;
15.. LET S21 (a) = S(2,1,a) ;
16.. LET S22 (a) = S(2,2,a) ;
17.. LET S23 (a) = S(2,3,a) ;
18..
19..//=================================================================
20..
21..#section conflictingbrake:
22..// ACHTUNG AB einer gewissen groesse HAENGT sf_car !! 1200 geht zB NICHT
23..// OK : SET brake = 800 ;
24..// OK: SET brake = WSPI("[0 800]") ;
25..// NICHT MEHR OK SET brake = WSPI("[0 800; 15 1500]") ;
26..// SET brake = 1200 ;
27..
28..SET brake = wspi("[0 800]") ;
29..CASE ANY EOF
30..
31..//=================================================================
32..#section startThenBrake:
33..SET  "throttle\nschedule" = wspi ("[0 60; 14.9 80;15 0;30 0]") ;
34..SET  "brake\nschedule"    = wspi  ("[0 0; 15.2 0; 15.3 500 ; 30 500]") ;
```

markuslepper.eu

```
35..CASE ANY EOF
36..
37..//CASE wspi ("throttle\nschedule","[0 60; 14.9 80;15 0;30 0]") > -10
38..//  && wspi ("brake\nschedule","[0 0; 15.2 0; 15.3 400 ; 30 400]") > -10 EOF
39..
40..//=================================================================
41..#section limits :
42..// speed is less than or equal to 110 mph, gear is less than 6,
43..// and acceleration is between -50 and + 50
44..CASE scope(2,1,vspeed) < 110 && scope(1,1,shift_logic/gear) < 6
45..  && scope(1,2,vsp') >= -50  && vsp' <= 50 EOF
46..
47..
48..//=================================================================
49..#section shift_to_next:
50..// each gear shift does maximally ONE step up or down
51..
52..LET geardiff = scope(1,1,gear - memory (gear)) ;
53..CASE abs(geardiff) <= 1 EOF
54..
55..
56..//=================================================================
57..#section shift_pauses_0.5, shift_pauses_2.0:
58..// no two adjacent gear shifts are closer than t seconds.
59..
60..LET shifting = S12( S11(gear) ~= memory(gear) ) ;
61..LET pause(t) = {OPT ~ shifting ; OPT REP {shifting ; MIN t ~ shifting} ;
62..                                 OPT {shifting ; ~ shifting} }
63..
64..#section shift_pauses_0.5:
65..CASE pause(0.5)  EOF
66..
67..#section shift_pauses_2.0:
68..CASE pause(2.0)  EOF
69..
70..
71..//=================================================================
72..#section reverse_shift_pauses :
73..// no two adjacent gear shifts in opposite direction closer than t seconds.
74..// t = 4 should pass, t = 7 should fail.
75..
76..LET upshift   = (gear > memory(gear)) ;
77..LET downshift = (gear < memory(gear)) ;
78..LET shift = upshift || downshift ;
79..
80..LET t = 7 ;
81..CASE REP { CASES REP {downshift ; ~shift} ;  MIN t ~shift
```


markusklepper.eu

```
82..           OR    REP {upshift ;   ~shift} ;  MIN t ~shift
83..           OR    ~shift
84..         } EOF
85..
86..//=================================================================
87..#section kickdown_downshift
88..// after a "kickdown" there has to be a downshift after
89..//    a given maximal delay of t1.
90..// t1 = 0.3 should pass, t1 = 0.001 should fail,
91..//    if test datas contains a kick-down.
92..
93..LET kickdown = diff(gas) > 100 ;
94..LET downshift = gear - memory(gear) < 0 ;
95..
96..LET t1 = 0.001 ;
97..CASE ~kickdown ; OPT REP {MAX t1 ANY ; downshift ; ~ kickdown } EOF
98..
99..
100.//=================================================================
101.#section kickdown_speed
102.// after a "kickdown" speed has to increase by at least 10 percent
103.//    after a maximal delay of t2.
104.// t2 = 1.5 should pass, t2 = 0.1 should fail,
105.//    if test datas contains a kick-down.
106.
107.LET t2 = 0.1 ;
108.LET kickdown = diff(gas) > 100 ;
109.LET kdspeed = shold(vspeed, kickdown) ;
110.CASE ~ kickdown ; OPT REP {MAX t2 ANY ; vspeed > kdspeed*1.10 ;
111.                           ~ kickdown }  EOF
112.
113.
114.//=================================================================
115.#section shiftdown_forspeed
116.// when shifting down WHILE ACCELERATING speed has to increase
117.//    by at least 10 percent after a maximal delay of t2
118.
119.// LET t2 = 0.1 ; // FAILS ! with brake pedal held : PASSES
120.LET t2 = 3.0  ;// PASSES ! with brake pedal held : PASSES
121.
122.// LET downfs  = (gear < memory(gear)) && vsp' > 1.0 ;
123.// GEHT NICHT HIER SIND JITTER ODER SO
124.// LET downfs  = (gear < memory(gear)) && vsp' > 5.0 ;
125.LET downfs  = gear < memory(gear) && delay( S11(vsp') ,0.2) > 1.0 ;
126.
127.LET shiftspeed = shold(vspeed, downfs) ;
128.CASE ~ downfs ; OPT REP {  MAX t2 ANY ; vspeed > shiftspeed*1.10 ;
```

```
129.                           ~ downfs  }
130.                OPT MAX t2 ANY  EOF
131.
132.
133.
134.//====================================================================
135.#section shiftup_forspeed
136.// when shifting up speed has to increase by at least p2 percent
137.// after a maximal delay of t2
138.
139.LET upshift = gear > memory (gear) ;
140.LET shiftspeed = shold (vspeed, upshift) ;
141.
142.LET pred(t2,p2) = {REP { ~ upshift ;
143.                         OPT { MAX 2.0 ANY ;
144.                               vspeed > shiftspeed * (100+p2) / 100 }
145.                       } }
146.CASES pred (2,12) AND pred (3,22) EOF
147.
148.
149.//====================================================================
150.#section brake_absolute :
151.// If there is a break force > b1 then in given time t1
152.// the speed has to be less than v1
153.
154.LET brktest (b1, t1, v1)  =
155.    { REP  { brake < b1 ;
156.             OPT {MAX t1 ANY ; vspeed < v1}
157.    }}
158.
159.CASES brktest (100,5,80) AND brktest (200,5,40) EOF
160.
161.//====================================================================
162.#section brake_percent :
163.// If there is a break force > b1 then in given time t1
164.// the speed has to have fallen with p1 percent
165.
166.LET brktest (b1, t1, p1)  =
167.     { LET cond = brake >= b1 ;
168.       LET brkspeed = shold (vspeed, cond) ;
169.       REP  {  ~cond ;
170.               OPT { MAX t1 ANY ; vspeed < brkspeed * (100-p1)/100 } }
171.     }
172.
173.CASES brktest (100,5,20) AND brktest (300,5,80) EOF
174.
```
End Of Source

markusslepper.eu

7.2 Explanations

7.2.1

This source file can be executed very easily: Just enter MATLAB and type

```
mwdemo (0)
```

at the command prompt. The model „`sf_car`“ will open, and so will a MWATCH Console linking this model to the source file.

Let's have a look at this source file from the beginning :

The first lines up to the first `#section` entry are common to all sections and contain some quasi „global“ definitions. In line 1/5 pp. just *abbreviations* are introduced for some important ports.

The block names containing a „newline“ character have to be enclosed in double quotes, so that the escape mechanism is applied. This is not necessary in line 1/8.

In line 1/9 a new signal named „vspeed'“ is generated by applying the built-in differentiation block to the signal „vspeed“.

Lines 1/11 introduce macro definitions with parameters; these are used to make the insertion of scopes into an expression less verbose. Here we also see that macro expansions can make further macro calls.

Let us skip the next two sections and continue with the first predicate, which can be found in line 1/41 in the section „`limits`“. Here just „instantaneos“ predicates are tested, i.e. there are not timing constraints and all predicates have to be fulfilled all the time.

Now you can acitvate the check box left to the name of the section and then press the button labeled „`Do Install`“. An MWATCH box will appear in the model and a scope will be opened which presents the signals selected in the source text. The color of the box should be „yellow“, indicating a launched but still „inconclusive verdict“. You can move the box in the model around or resize it as you like. You can even delete it manually and see all `goto` blocks being deleted, too, automatically.

If you double click the box the mask will pop up and you can e.g. activate the abortion of the session as soon as the predicate fails by activating the check box „`Stop on Failure`“. If you select the box and select „`E dit -> Look U nder Mask`“ from the models menu you see the network into which the source code predicates are compiled. Feel free to move and resize the primitive blocks contained herein, but please do not delete connections or alter names. Normally you will never need to look at these compiled devices, but it may be fun having a look behind the scene.

Now start the simulation as usual and watch the box changing its color. After the simulation run press „`get Verdicts`“ in the MWATCH Console and the verdicts will be copied and the colors of the name entry fields will change accordingly.

At last you can press the button „`Write Protocol`“ and you will find your activities eternalized in a littel XML protocol in the file

<*mwatchdir*>`/demo/sf_car_mw_results.xml`

markuslepper.eu

The next predicate „shift_to_next“ also works without temporal constructs, but some timed specification is realized by the „memory()“ construct, which uses a Simulink block for comparing the current value of gear with that of the last simulation step, thus detecting gear shift events.

Nevertheless this predicate is still instantanuos from the ///ATCH point of view. It simply says that the difference between the value of gear in two adjacent steps must not be greater than one or less than minus one. One scope with one pane and two channels is installed using the short-hand notation we introduced in the common prefix of the file.

7.2.2

The predicates contained in the sections shift_pauses_⟨*duration*⟩ are more interesting : They macro pause(t) specifies that between two gear shifts there has to be a pause of t time units (seconds). The sets of traces fulfilling this macro have to be constructed as follows :

- At the beginning there may be an arbitrarily long interval where no gear shift occurs („OPT ~ shifting“, please notice the logical „not“ operator „~“).
  The shifting is an instantaneous predicate defined in line 1/60, using the matlab comparison operator „~=“, which is packed into a simulink block. This generates a boolean signal from two numeric signals.
- Then a gear shift may occur, followed by an intervall of t seconds, in which no shift occurs.

Please notice that a macro defining a set of traces has to enclose its body in curly braces. Also the argument of REP, which must be a sequence of predicates combined by the chop operator ";" must be enclosed in curly braces.

The following two sections simply instantiate this macro with two different durations. If you activate these predicate and start the simulation the first one will succeed and the second one will fail.

Please note that this predicate is only fulfilled, if *at the end of the simulation* there is also a pause of t after the last gear shift. If this is not intended, because of the nature of the segment of test data, and if you want this test data even to end with a gear shift, the predicate has to be altered like ...

Source Example 2

```
1... LET pause (t) = { OPT ~ shifting ;
2...                   OPT REP {shifting ; MIN t ~ shifting } ;
3...                   OPT shifting ; OPT ~ shifting }
```

End Of Source

Now the *last* gear shift can be followed by an arbitrarily short interval of non-shift, or even happen in the very last trace sample.

The next predicate in section reverse_shift_pauses is a bit more special as it compares gear shifts in opposite directions. The CASES combinator is used to combine the three possibilities : Either no shift occurs at all (last case) or one or more down shifts occur, followed by an interval of t seconds, in which no up shift occurs, or vice versa.

markusslepper.eu

The predicate kickdown_downshift beginning in line 1/87 first defines a cick-down event by comparing the derivative of the throttle value with a certain threshold, generating a boolean signal kickdown.

The temporal predicate has be be read as:

At the beginning there is no kick-down-event. Optionally this is followed by a repeated sequence of one or more kick-down events followed by a downshift. The sequence of kick-downs is limited to be of maximal t1 length.

Here we see a typical „idiom“ of these kinds of trace semantics: The kick-down event(s) are not specified explicitely, but by an implied negation. Since all subtraces described by „∼ kickdown“ must end as soon as kickdown gets true, the following time segment *must* be interpreted as corresponding to the ANY segment ! In this subtrace there may be arbitrarily many changes of kickdownbecoming true and false, but the first true phase starts the timer and requires the downshift to follow after maximal t1 time units.

The section kickdown_speed has the same structure. In the „deterministic“ part of the specification, i.e. in the formula which are evaluated independently from their temporal context, some time-orientatin is introduced by the „sample-and-hold“ device. This is a block containted in the MWatchLib blockset and holds its first input value as long as its second input is false and samples the current value if the second input is true. So we memorize the speed valid at the time of the *last* kickdown and propose that after t2 time units speed has to have increased by 10 percent.

Please note that expressions like „memory()“ and „sandh()“ are realized in the static part, i.e. not reentrant, so that we can not refer to the vspeed of the *first* kickdown parsed by the temporal formula, – this parsing is non-deterministic and instantiated arbitrarily often, but there is only one network realizing the instantat-neous predicates.

The next predicate shiftdown_forspeed looks for the event of a downshift together with increasing speed of the vehicle. Please note that we have to use a short delay of the vspeed' signal for *technical reasons*: The evaluation mechanism of simulink seems to produce some „random“ noise when calculating hard changes of the throttle value which can be seen nicely on the scope installed by this predicate.

7.2.3

The next predicates all have similar structure. The last predicate demonstrates the use of *local macro definitions* and *local instantiations* of non-reentrant blocks like shold(). As mentioned above these are only installed once and cannot be evaluated non-deterministically, but of course they will be installed n times and operate independently, if the macro which installs them is instantiated n times, as it is the case in our example.

7.2.4

The first two sections demonstrate the *generation* of test data:

In section conflictingBrake the value of the brake signal is reposed by our definition (a simulink „from Workspace“ block delivering a constant value of 800), but the signal „throttle schedule“ remains untouched, so that a situation is simulated where gas and brake are pushed simultanuosly.

markusslepper.eu

please note : If the value for brake gets too large, this simulink model simply **hangs** :-(

In section startThenBrake both signals are replaced in a more senseful manner: A start and a full brake are simulated.

Please combine *one* of these predicates with one or more of the others to see if the verdicts change accordingly. Of course predicates specifying the behavior of the brake system can only work if the brake is used in the test data.

please note : When a test data is defined for a outport which is not connected at all this is considered an error and the corresponding MWATCH block will not successfully be launched. This will be the case if *both* of the first predicates should be tried to install. – the first will install successfully, the second then will fail, because the port it wants to redefine is yet disconnected.

A Embedding into MTest

This section addresses users familiar with the MTest test environment by DC.

When running MTest and you have sucessfully

1. selected a model to test,
2. selected a subsystem to test,
3. selected a test sequence,
4. selected a singular test,
5. and opened (or created) the test bed,

then the menu entry

TestBed / Edit Watchdogs

will launch a MWATCH Console for the system under test.

preliminary : Specification files are related to the single test, not to a test suite or a model.

If such a specification file *already exists* for the current test (i.e. the file exists in the *directory* containing the test data), the MWATCH Console is launched with this file.

If such a file *does not exist*, a template file will be created automatically containing shortcut macro definitions for all visible ⟨*TestPoint*⟩s in the test bed. This is done by calling the API function mwMaketemplate() from table 2 above.

If you want to reuse an existing predicate file from one testcase for another one, you have to copy it *manually* between the respective directories and then maybe adjust the text to your needs.

preliminary : ATTENTION the MWATCH Console will not disappear automatically if the selection of test, test suite, subsystem or model is changed, but all further operation on this console will result in some error. Simply close the console window and activate the menu entry TestBed / Edit Watchdogs again to get a new MWATCH Console for the new setting.

markusslepper.eu