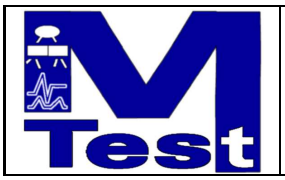


Model-based Testing for Matlab/Simulink	Technical Notice TN MTest-2001-5

MMATCH Extended Tutorial

Markus Lepper
— ÜBB/TU Berlin —

Contents

List of Figures

List of Tables

1 Introduction to the MONITOR Principles

[0] The following text is an attempt of a small interactive tutorial, introducing several aspects of the MONITOR tool.

While trying to abstract from all theoretical issues, the author assumes that some introduction into the underlying principles of the architecture can be helpful.

[1] From a systematic point of view there are clear distinctions between the different layers of

- the MONITOR algorithm,
- the MONITOR tool,
- and the MONITOR language, – which itself consists of two(2) layers.

But since this paper tries to describe the operation of the tool from the practical viewpoint, we mostly will neglect these distinctions, recurring to them only where unavoidable.

- The kernel of MONITOR is an *algorithm* for real-time evaluation of the conformance of a system's behavior with a specification made up from temporal properties.
- This specification is given using the MONITOR *language*. The MONITOR language *directly* represents the trace semantics evaluated by the algorithm, i.e. works without quantors or the necessity of declaring variables.
- The existing *tool implemetation* supports a direct simulation time interface to the MATLAB/simulink simulation environment for evaluating specifications, a GUI for interactive selection of specifications, and an API for program controled operation[1].

[2] Since the current implementation of the tool works with MATLAB/simulink, for sake of the user's convenience the lower level of the MONITOR language (containing the so called „instantanuos predicates“, see below [41] *pp.* on page 13) follows closely the syntax and semantics of the corresponding MATLAB und simulink operations[2].

The upper level of the language, containing the syntactic constructs for combining the instantanuos predicates to temporal trace specifications, is independent from this underlying expression language.

[1] Currently we support MATLAB 5.3.1 and simulink 3.0.1 (that is Release 11.1)
[2] . . . which themselves are not always consistent w.r.t. each other!

2 Basic language constructs

[3] Each MWATCH specification can be read und understood as a kind of „regular expression“, which is well known from automata construction: Each specification describes a „family“ of traces, i.e. a set of simulation runs, which fulfill the specification.

The similarity to regular expressions comes from the fact that the specifications describe the run of the system dia-chronuosly: The leftmost part of a (sequential) MWATCH formula describes the start of the system run, the next expressions describe the next segments of realtime, etc., – the rightmost expression describes the required behavior at the end of each run.

[4] Also well known from regular expressions is the possibility of disjunction, i.e. the specification of `OR`-connected *alternatives*.

[5] Not found in regular expressions is the `AND` conjunction, i.e. the requirement for a run to fulfill several predicates simultanuosly.

[6] Each MWATCH specification contains three layers of syntax and semantics:

1. Firstly the signals and output ports which shall be covered by the specification (and thus observed during simulation time) have to be identified.
2. Then „instantanuos predicates“ are formulated, which must be met by the current values of these signals and ports in certain *time instances* of a simulation run.
3. At last these instantanuos predicates are combined to a temporal sequence which must be met by the system's behavior throughout the duration of a complete simulation run.

[7] For illustration here is an **example** with an indication of these three levels. This example will be explained in detail in the following.

```
CASE   shift_logic/gear  <=  3   ;   shift_logic/gear  >  3   EOF
```

2.1 Applying MWATCH Specifications to simulink Systems

[8] In the current implementation an MWATCH specification is applied to a simulink model by (1) installing an MWATCH function block into a simulink system, and (2) identifying a specification for this block.

9 Whenever a simulation of a model is run, all MWATCH blocks which currently exist in this model perform the conformance test between the model's behavior and the selected specification automatically.

10 The installation of an MWATCH function block can be done program controled or manually (by drag n' drop out of the library). The further way of operation however is identical.

11 An MWATCH function block can be inserted also in subsystems of arbitrarily deep nesting level, if they are *not realized by unbroken library links*.

12 The system *immediately* containing a certain MWATCH function block is called the corresponding „Sub-System Under Test“, or SSUT in the following.

13 This name comes from the fact that each MWATCH function block can observe the sequence of values on all signals and output ports which are contained in its SSUT, either immediately, or in a subsystem of arbitrarily deep nesting level.

2.2 Selecting Signal Values (PTOs)

14 As mentioned above (see 6) the first decision when writing an MWATCH specification is the identification of the signals which shall be subject to observation.

15 In the current implementation the values of *named signals* and of *output ports* can be observed. Together they are called „Points of Test and Observation“, or PTOs, in the following[3].

16 Currently there is the **restriction** that only PTOs containing **single** values can be observed, i.e. there is **no access to matrix valued or complex** signals.

17 PTOs are adressed in an MWATCH predicate by giving their complete path names *relatively* to the SSUT(=the immediately containing system) of the MWATCH block which is going to interpret the predicate.

18 As well-known from the simulink-API, a pathname of a port or signal consists of components, all seperated by slashes

/

with *no intervening* blanks.

19 As known from simulink, **all components of a PTO's path name are case sensitive.**

20 The first components of the path name are the names of the subsystems containing the PTO, from outermost to innermost. If the PTO is contained in the SSUT

[3]These are standard terms from the field of Protocol Conformance Testing, and we consider them quite adequate.

immediately, this sequence of names is just empty.

21 This prefix is followed either

1. by the name of a signal,
2. or by the name of a block and the name of the selected output port,
3. or by the name of a block and a small integer number, giving the number of the output connector,
4. or by the name of a block whithout further specification, meaning the one and only output of this block.

These components of the path name are also seperated by slashes.

The last method of selecting a PTO is of course only applicable if the block indeed has only one single output.

The last two methods of selecting a PTO will mostly be used when observing one of the built-in simulink function blocks, which mostly do not assign port names to their outputs.

22 If all of the components of a path name are made up *only* from alphanumeric characters and the underline character "_", the whole pathname corresponds to a single *identifier* in the sense of the MATCH lexis and can be written directly in a predicate's text.

23 If one of the components of a pathname contains special characters (like "+","?", "!"), blanks or even newline characters, the *whole path* must be included in double quotes

"..."

24 With this second notation, *all* characters included in the double quotes are considered part of the name, including leading, intervening and trailing blank characters.

Furthermore the sequence

`\n`

can *not* be part of an identifier, but is interpreted as a *newline* character, which is contained in the names of some predefined simulink function blocks[4].

[4] **Please note** that some of the predefined simulink library blocks (perfidiously) contain blank characters *adjacent* to newline characters. These blocks can only be adressed by finding out their „invisible name". From the GUI this can only be done by activating the „change name" function and stepping with the cursor through the characters!

Table 1 Examples for valid path names of PTOs, if `sf_car.mdl` is the SSUT.

1. By the name of a signal:
   `"vehicle\nspeed"`
   `"transmission/turbine torque"`

2. By the names of a block and of the selected output port:
   `Engine/Ne`
   `"transmission/transmission\nration/Tout"`

3. By the name of a block and a small integer number:
   `Engine/Sum/1`
   `Vehicle/mph/1`

4. By the name of a block whithout further specification:
   `Engine/Sum`
   `Vehicle/mph`
   `"Engine/engine + impeller\ninertia"`

2.3 Using this tutorial and the MWATCH console

25 This tutorial refers to the example model

`sf_car.mdl`

contained in each MATLAB/simulink distribution.

After installation of MWATCH[5], just type

`MWatchTutorial`

at the MATLAB prompt, and this model will open, together with an MWATCH console[6].

Furthermore the test input data (`brake\nschedule` and `throttle\nschedule`) are adjusted for better fitting to our example specifications.

26 Table 1 shows some valid path names for the case that `sf_car.mdl` is the SSUT, ordered by the categories of supported forms as given in 21.

27 The MWATCH **console** popped up when typing "`MWatchTutorial`" combines the selected model "`sf_car.mdl`" with the specification file

<mwatchdir>`/demo/mwtutorial.mw`

[5] See Installation and User's Guide.

[6] In contrast to simulink function block and system names, commands in MATLAB are **not case sensitive**.

which contains the texts of the following examples.

28 In the current implementation all MWATCH specifications must be contained in a *disk file*. This file is divided into *sections*, each section corresponding to one specification. The MWATCH console displays one row for each section name contained in the file, preserving their textual order[7]. Each such line contains a check box, the section name, and the *verdict* delivered by the specification the last time it had been evaluated.

29 Please activate the small checkbox beside the predicate name "`p00`", then click onto the "`Do Install`" button.

An MWATCH function block assigned to the evaluation of the file section "`p00`" will be installed, together with a new `scope` object with four panes.

This installation of a corresponding MWATCH function block is called *launching* of the predicate.

The new MWATCH function block thus corresponds to the text contained in the text file as it is *at the moment* of launching.

30 ⊕⊕⊕ Launching of a specification will also be triggered whenever you activate the "`compile and load`" button of the MWATCH function block's mask, see Installation and User's Guide.

31 ⊕⊕⊕ Whenever a predicate is launched the following sequence is performed:

1. The MWATCH compiler searches the file with the given name
2. The section with the given name is compiled into a temporary file named "`MWATCHcode.m`".
3. This file is executed for installing the internal signal processing network and all goto- and from-blocks, as well as initializing the MWATCH function block's mask with the serialized version of the specification.

32 ⊕⊕⊕ The directory where to put and find the temporary code file is determined by a value of the mask of the configuration block in the MWATCH library. If you note strange effects on your specification, there may be an old file named "`MWATCHcode.m`" from another directory, which appears with higher priority in the built-in MATLAB search path.

33 Now, please start the simulation, and resize and rescale the scope after the simulation has finished.

34 If you click onto the "`Edit Specs`" button, the MATLAB editor will be invoked on the specification file. Look for the line

```
#section p00 :
```

[7]Since each section name can occur multiple times in a file, the corresponding text sections being concatenated to form the specification, this sequence of section names actually reflects their *first* appearance in the file, see Installation and User's Guide.

and after this you will find the code which is responsible for displaying the value history of some PTOs on the scope's panels.

Please ignore all the syntactic constructs in this section except the path names of the PTOs.

35 For each PTO which is mentioned in the specification, a simulink „goto-block“ has been installed automatically. These are placed in a constant distance from the corresponding output connector. If you open the corresponding subsystems you will find the goto-blocks of the PTOs of deeper nesting level.

36 If you mark the MWATCH function block and select the "`Edit⇒Clear`" menu function (or simply press the delete-key) the function block will be deleted, as expected. But also all goto-blocks related to *this* MWATCH function block will be deleted automatically.

Please do not change the name or the tag value of such automatically created goto-blocks, since this automatic deletion will not work any longer. But you can always move them around, if appropriate, – maybe you detect a second one hidden under a younger colleague, since two distinct MWATCH function blocks observing the same PTO will install two distinct goto-blocks.

37 ⊕⊕⊕ All goto- and from-blocks installed by MWATCH have a „tag value“ starting with the character sequence "`MWATCHTAG`". Please be sure that no other block uses such a tag value, – the results may be unpredictable.

38 If you activate the "`Do Install`" function of a console, all MWATCH function blocks which had been installed by this console and do still exist, will be deleted, with the same consequences as done manually. Afterwards the specifications with a marked check-box are installed „from scratch“. Therefore you always may delete MWATCH function blocks manually without confusing the console.

39 **Feel free** to edit this section of the file[8] by replacing the path names of the PTOs by your own choices, but please do not alter the other syntactic components of this section.

If you now *save the file* and press "`Do Install`" again, either the altered file is executed and you see the selected values on the scope's panel when running the simulation again, – or you get an error message, because there are typos in the new path names.

40 **Please note** that *after each editing* of a specification file you must always (1) save the file to disk explicitely and (2) press "`Do Install`" again. **The MWATCH tool will not be notified on alterations of already launched specification files automatically.**

[8] . . . as long as you can reconstruct it, – e.g. from the distribution `;-)`

Table 2 Basic arithmetic and logic operations

Taking arithmetic values (int or float) and yielding a new float value:

Binding power: equal decreasing

e_1	`*`	e_2	Multiplication
e_1	`/`	e_2	Division
e_1	`+`	e_2	Addition
e_1	`-`	e_2	Subtraction

Taking arithmetic values and yielding a logical value:

e_1	`<`	e_2	Less-than comparison
e_1	`<=`	e_2	Less-than-or-equal comparison
e_1	`>`	e_2	Greater-than comparison
e_1	`>=`	e_2	Greater-than-or-equal comparison
e_1	`=`	e_2	Comparison for equality
e_1	`~=`	e_2	Comparison for inequality

Taking logical values and yielding a new logical value:

	`~`	p	Negation of a predicate
p_1	`&&`	p_2	Konjunction of two predicates („and“)
p_1	`!!`	p_2	Disjunction of two predicates („or“)
p_1	`=>`	p_2	Implication
p_1	`<=>`	p_2	Equivalence

Table 3 Commonly Used Symbols for Syntactic Categories

e	arithmetic *E*xpression
p	instantanuos *P*redicate
d	*D*uration constant (float or integer)
i	*I*dentifier
n	integer constant
s	complex *S*ubformula

2.4 Instantanuos Predicates

41 The most simple form of predicates which can be checked by MWATCH are called *instantanuos*. These predicates refer to a „snapshot“ of the SUT, and make propositions on the values of ports and signals found in such a snapshot.

No notion of time is involved in the formulation of such predicates.

Instantanuos predicates are the main building block, on top of which which all more complicated MWATCH predicates are constructed.

42 Instantanuos predicates can be thought of as „signals of type boolean“. This signal has to be „true“ for the predicate to be fulfilled by a certain combination of values.

In the following we will represent an instantanuos predicate by the symbol

43 A PTO which already has the simulink-type „boolean“ can *directly* be used as an instantanuos predicate. This signal must be `true` for the corresponding predicate to be fulfilled.

44 All PTOs with the simulink-type integer or float are treated uniformally: They can be used in *arithmetic expressions*, deriving new values from the current values of the PTOs[9].

45 From all numeric values (i.e. immediately from PTO values or from values derived from those by arithmetic expressions, or from numeric *constants*) a boolean value must be derived by applying a *relational operator* to two of them.

46 From these boolean values, together with PTO values which *sui generi* are of simulink-type boolean, further boolean expressions may be derived by applying *logical operators*.

Table 2 lists all operators of these three categories. Please note that the „wording“ of these operators is determined by the usage in MATLAB/simulink.

47 ⊕⊕⊕ **Please note** that the MWATCH compiler has *no information* concerning the types of the PTOs as defined implicitely by the kind of function blocks they emerge from.

Due to the somehow „historically grown“ architecture of simulink, there is not even a type checking when a new connection is established. Typing errors will be notified to the user **not before the simulation does start**.

Sorry for this, but this is due to simulink's idionsyncratics. :-(

[9] Furthermore there are some MATLAB/simulink *functions* which are accessible from the MWATCH language. These will be introduced later, see 2.16.

48 Concerning our example patch „`sf_car.mdl`“ such an instantanuos predicate could be

„The number of the current gear is less than or equal to three(3)“

which can be a meaningful predicate for certain test cases.

But also it is possible to postulate...

„The number of the current gear is greater than the current speed
divided by the throttle percentage plus fortyseven(47)“

which will not make too much sense.

49 In our example (`sf_car.mdl`) the currently selected gear is indicated by the value of the output "`gear`" of the subsystem "`shift_logic`".

The predicate mentioned above can therefore be formulated as ...

```
shift_logic/gear <= 3
```

50 The second predicate from above would be written

```
shift_logic/gear > "vehicle\nspeed" / ("throttle\nschedule" + 47.0)
```

2.5 Invariant Properties

51 The most simple form of a temporal expression in the MWATCH language consists only of *one single* instantanuos predicate. It has the meaning, that this instantanuos predicate must be fulfilled by *all time instances* of any given run which matches this specification.

The syntactic construct in the MWATCH language for „lifting“ an instantanuos predicate p to be a temporal formula is simply

```
CASE   p   EOF
```

52 Writing

```
CASE shift_logic/gear <= 3 EOF
```

is thus a complete MWATCH specification, saying that in each instance of a run the current gear must be less than or equal to three.

Such a specification can be accounted as the most simple form of a „temporal predicate“, – indeed it is technically treated as such, – nevertheless it is nothing more

than an „invariant“ of the SSUT, because the required property does not change with time.

53 **Please note** that *all* MWATCH keywords are *case sensitive*, – if you happen to delete the "`CASE`" when editing this example, please do not reconstruct it to "`Case`" or "`case`".

54 Please go back to the MWATCH console and deactivate all checkboxes and activate the checkbox next to "`p01`". No click "`Do Install`" again and start the simulation. As soon as the gear will raise above three(3) the specification is not fulfillable any more and the MWATCH function block will turn **red**.

55 ⊕⊕⊕ If you mark the MWATCH function block and then select the simulink menu function "`Edit/LookUnderMask`" (or simply press ctrl-U), the signal processing network constructed by the MWATCH compiler for realizing your specification will become visible.

From time to time it could perhapes be useful to have a look at this network, or even place a „floating scope“ somewhere inside. If you even edit it, please **do not remove from- or goto-blocks**, because the automatic deletion of the corresponding goto- and from-blocks will not work any more when deleting the containing MWATCH function block.

56 If you click on "`Edit Specs`", the editor will pop up again and you can alter the predicate which follows the line `#section p01` as you like. If you change the numeric constant, the specification will fail more or less early when running the simulation.

This is caused by the curve of the "`gear`" signal as induced by the default input test data. If you use predicate "`p02`" instead of "`p01`", additionally a scope is installed which shows the curve of "`shift_logic/gear`".

57 Be reminded that after each alteration to the specification text you must relaunch the corresponding MWATCH function block explicitely, as described above in 40.

58 ⊕⊕⊕ Every single MWATCH function block has the ability to *influence* the run of a simulation by aborting it, in case of conformance as well as in case of failure:

- If you double-click on the MWATCH function block, a simulink-style *mask* should pop up for this block.
- Then activate the checkbox in the near to "`Stop on Failure ?`",
- and click "`OK`" (the mask vanishes) or "`Apply`"(the mask stays open).

When starting the simulation again, the failure of the specification will *stop* the simulation run, popping up an error message window.

59 Please replace the integer constant in `section p01` with a value high enough, e.g.

```
CASE shift_logic/gear <= 100 EOF
```

This specification will be *fulfilled* at the *end* of each simulation run, and the MWATCH function block will finally turn **green**.

This „passed“ verdict however will not be fixed until the end of the simulation run. This is (of course) because the trace data can fail to fulfill the invariant even in the very last moment of the simulation run.

2.6 Simple Temporal Properties, Sequentialization

60 The invariant properties considered in the last section contained only one instantanuos predicate, which had to be fulfilled by all instances of the *whole* simulation run.

Now we introduce the first really „temporal“ specification by simply dividing the trace of the whole run into *subtraces*, each of which is characterized by an instantanuos predicate of its own.

To express the sequentialization of conditions, the central operator is the „chop“ operator, denotated by a semi-colon

```
;
```

and sequences of instantanuos predicates are written using the pattern

CASE p_1 ; p_2 ; ... ; p_2 EOF

61 If we want to express that the simulation run of our SSUT must *start* with a gear less than or equal to three, but then may behave arbitrarily, we could write

```
CASE shift_logic/gear <= 3 ; 1 < 2 EOF
```

Since 1<2 is always true, this MWATCH predicate expresses exactly what we want: The whole run has to begin with a sub-section in which `shift_logic/gear <= 3` holds, after that the system may behave arbitrarily, since the truth of $1 < 2$ does not depend on its behavior, – fortunately `;-)`

62 The wording of this predicate is definitive correct, but things like „1<2“ will probably be confusing to the reader. So we better use the special keyword

```
ANY
```

which stands for *each arbitrary* behavior of a system. So now we can better write

```
CASE shift_logic/gear <= 3 ; ANY EOF
```

which is semantically the same as the property above in 61.

63 **Please note** that (as mentioned above in 53) all MWATCH keywords are *case sensitive*, – writing "`any`" or "`Any`" will let the tool search for a signal with this name and does not denote the special keyword.

64 Please alter section "`p01`" (or "`p02`") accordingly (by inserting the character sequence "`; ANY`"), re-install the corresponding MWATCH function block (by activating the checkbox and clicking "`Do Install`" in the console), and run the simulation. The MWATCH function block will turn green *almost immediately*, because the positive verdict is fixed as soon as the final `ANY` is reached, – which is the case immediately after the leading "`shift_logic/gear<=3`" has been verified for the first simulation step of the system.

65 The time instance in which the tool fixes the „verdict“ on the simulation can be seen by clicking onto the "`Get Verdicts`" button in the console. Doing this all verdicts of all predicates for wich an MWATCH function block is currently installed will be copied into the corresponding line of the console's display.

66 ⊕⊕⊕ If you „look under the mask“ of the MWATCH function block again, as explained in 55 above, you will notice that the „patch“ built up by the MWATCH compiler is the same as before.

Indeed only the evaluation of the basic instantanuos predicates contained in a specification (i.e. the „lower level“ of the MWATCH language) is delegated to simulink, by constructing the appropriate signal processing network and generating boolean signals.

The logic (and secret ;-) of MWATCH is contained in the `.DLL` which realizes the nondeterministic evaluation of the „higher level“ of the MWATCH language, operating only with functions from time to boolean as inputs.

67 Of course the syntactic elements introduced so far can be used with identical semantics in arbitrary combination, – a property which is called „compositionality“.

Please make some experiments like

```
CASE ANY ; shift_logic/gear <= 3 EOF
```

which succeeds if the trace *ends* with a sub-segment in which the gear value is ≤ 3, independent from the precedent behavior of the system.

68 Also a typical pattern is presented by the specification

```
CASE ANY ; shift_logic/gear <= 3 ; ANY EOF
```

which requires, that *sometimes*, – at an arbitrarily chosen time instant, – there starts a non-empty segment which fulfills the predicate `gear`≤ 3. So this pattern

corresponds to the „eventually“ operator ($\Diamond$) from classical temporal logics, – while simply writing `CASE` p `EOF` corresponds to the „always“ operator ($\Box$).

69 Please verify that

```
CASE ANY ; shift_logic/gear > 3 ; ANY EOF
```

succeeds as well, fixing the verdict as soon as possible.

Whereas

```
CASE ANY ; shift_logic/gear > 4 ; ANY EOF
```

does *fail*, but not until the end of the simulation: The SSUT is given the chance to succeed up to the very last moment of its run:

Up to the very last moment the trace data could still belong to the first `ANY` segment, reaching the middle and last segment with the last sample step of the simulation run.

70 Here is another typical pattern of an MATCH specification:

If you want to express that the gear may be larger than three, but only in a middle segment of the whole run, and only for a limited period of time, you may write :

```
CASE shift_logic/gear <= 3 ; ANY ; shift_logic/gear <= 3 EOF
```

This example demonstrates a typical style of writing with MATCH: The desired behavior is specified by a kind of *negation*.

Since the keyword `ANY` stands for an *arbitrary* behavior of the SSUT and since we claim for the gear being ≤ 3 in the first and last segment, only the middle segment would allow a gear larger than 3, because it does *not* impose this restriction.

71 The above specification *allows* the SSUT to change the gear to a value larger than three(3), but does not enforce this behavior, – the gear may stay low, if the SSUT wishes so.

Futhermore, the SSUT may switch to and fro a larger gear arbitrarily often, as long as all sub-traces where the gear is large together fit into this free middle segment.

If we want to express that the gear *must* cross this limit, and must do this exactly *once* , we have to write a far more restrictive specification:

```
CASE  shift_logic/gear <= 3 ; shift_logic/gear > 3 ;
      shift_logic/gear <= 3    EOF
```

72 The formulation for switching the gear *at least* once over this limit, but otherwise aribtrarily often in the middle of the run is ...

```
CASE  shift_logic/gear <= 3 ; shift_logic/gear > 3 ; ANY ;
      shift_logic/gear <= 3    EOF
```

73 ⊕⊕⊕ If you look now install predicate "p03", watch the scope displaying the trace of the gear value and compare it to the time instant when

```
CASE  ANY ; shift_logic/gear > 0 ; ANY ;  shift_logic/gear = 1  ;
      shift_logic/gear = 2 ; ANY
EOF
```

becomes true.

This will happen *as soon* as the gear value only „touches" the value 2. The MWATCH algorithm always assumes that – if a value required by an instantanuos predicate is reached in only one time instance, there is *always* a small interval of time „around" this instance in which the predicate holds throughout. Because the rest of the specification only requires "ANY", the whole specification is considered as fulfilled immediately.

74 ⊕⊕⊕ Furthermore we see that the time intervall in which the gear actually has the value 1 is „used" for *four different segments*, for the initial ANY-segment, for the condition gear> 0, for the next ANY-segment, and for the segment requiring gear=1.

The MWATCH semantics require for the chop operator ";" only, that there *exists at least one* segmentation of the actual trace of values. The question, „which of these segmentations has actually been chosen by MWATCH" cannot be answered, – the MWATCH algorithm calculates the existance of one or more possible segmentations, and does not make such a decision, which indeed would be totally arbitrary w.r.t. the semantics.

2.7 Simple Macros (without Parameters) used as Abbreviations

75 With the predicate of 71 it is somehow inconvenient and error-prone that the same predicate must be typed three times, – twice positve, once negated.

So the „macro definition" feature of the MWATCH compiler can be applied here with some benefit to abstract a complex condition (or an arithmetic expression) into one single identifier.

The syntactic patterns are

```
LET  i  =  p ;
LET  i  =  e ;
```

where i stands for an arbitrary identifier chosen by the user, which should be selected not to „shadow“ the name of any accessed PTO (or needed built-in function, see below 2.16).

The symbols p and e shall indicate that you can abstract from instantanuos predicates as well as from arithmetic expressions.

Together with the operator for logical negation „~“ this version of 71 is much more readable:

```
LET  lowgear = shift_logic/gear <= 3 ;
CASE lowgear ; ~ lowgear ; lowgear EOF
```

2.8 Specifying Minimal Durations of Sub-Traces and of the Validity of Predicates

76 Of course with temporal specifications a central deserve is to specify the *durations* a certain property (1) must minimally be fulfilled, or (2) may maximally be used for defining a subtrace.

For this sake the MATCH language provides the syntactic constructs

```
MIN  d   p
MAX  d   p
```

Here p indicates one single instantanuos predicate[10], and d stands for a numeric constant[11], giving a duration in *seconds*.

The `MAX` prefix will be discussed in the next section.

77 The `MIN` prefix assigns a minimal duration value to a subtrace. Only these segmentations of the total trace are valid, in which the sub-trace corresponding to this sub-formula has the required length.

If this sub-formula is an instantanuos predicate p (as it is in all examples of this section) the proposition that the corresponding subtrace has a length of minimally d *implies*, that this predicate is valid for at least the given duration.

Consider (and execute!) the following specification:

```
CASE MIN 2.0 shift_logic/gear <= 3 ;  ANY  EOF
```

[10] As we will see later, *all* constructions, e.g. sequences of subtraces combined with the chop operator "`;`" and even more complex ones, may be prefixed by a `MIN`/ `MAX` specification.

[11] No expressions supported here at the time, sorry `;-(`

This specification requires that the whole trace can be divided in two subtraces, such that in the first the given predicate holds, and that this first trace lasts at least 2.0 seconds.

Positively spoken: After the start of the simulation the value of `gear` must be ≤ 3 for at least 2.0 seconds.

In the typical way of „negative formulation“ this also means, that „not earlier than 2.0 seconds after system start the gear may be larger than three“.

78 When using specification `p02` and watching the curve of the `gear` value, it is easy to see that the specification

```
CASE MIN  10.0 shift_logic/gear <= 3 ;  ANY  EOF
```

will **fail**.

79 Because of this „negative meaning“ the consequences of a `MIN` prefixed formula therefore also may be related *not* to the prefixed expression, but to the *following* subtrace:

```
CASE MIN 10.0 ANY ; shift_logic/gear > 3 ;  ANY  EOF
```

Here the „positive“ formulation hardly makes sense: „For the first 10.0 seconds *or longer* the system may behave as it likes to, and then the gear must be larger than three, – followed by any behavior.“

We hope the true meaning of this formula becomes clear when switch to the „model theoretic“ level, i.e. treat the formula as a regular expression and look at the possible sequences of subtraces it describes (can produce / can consume) :

The trace data fulfills this specification **if and only if** it is **possible** to split the total trace into three segments, the first of which must be last at least 10.0 seconds, the second must have a `gear` value larger 3, and the last is arbitrary.

With this specification *three* situations can arise, depending on the trace data, which are depicted in figure 1.

The effect of the specification could be re-formulated as „Sometimes after the first 10.0 seconds the gear must be > 3“.

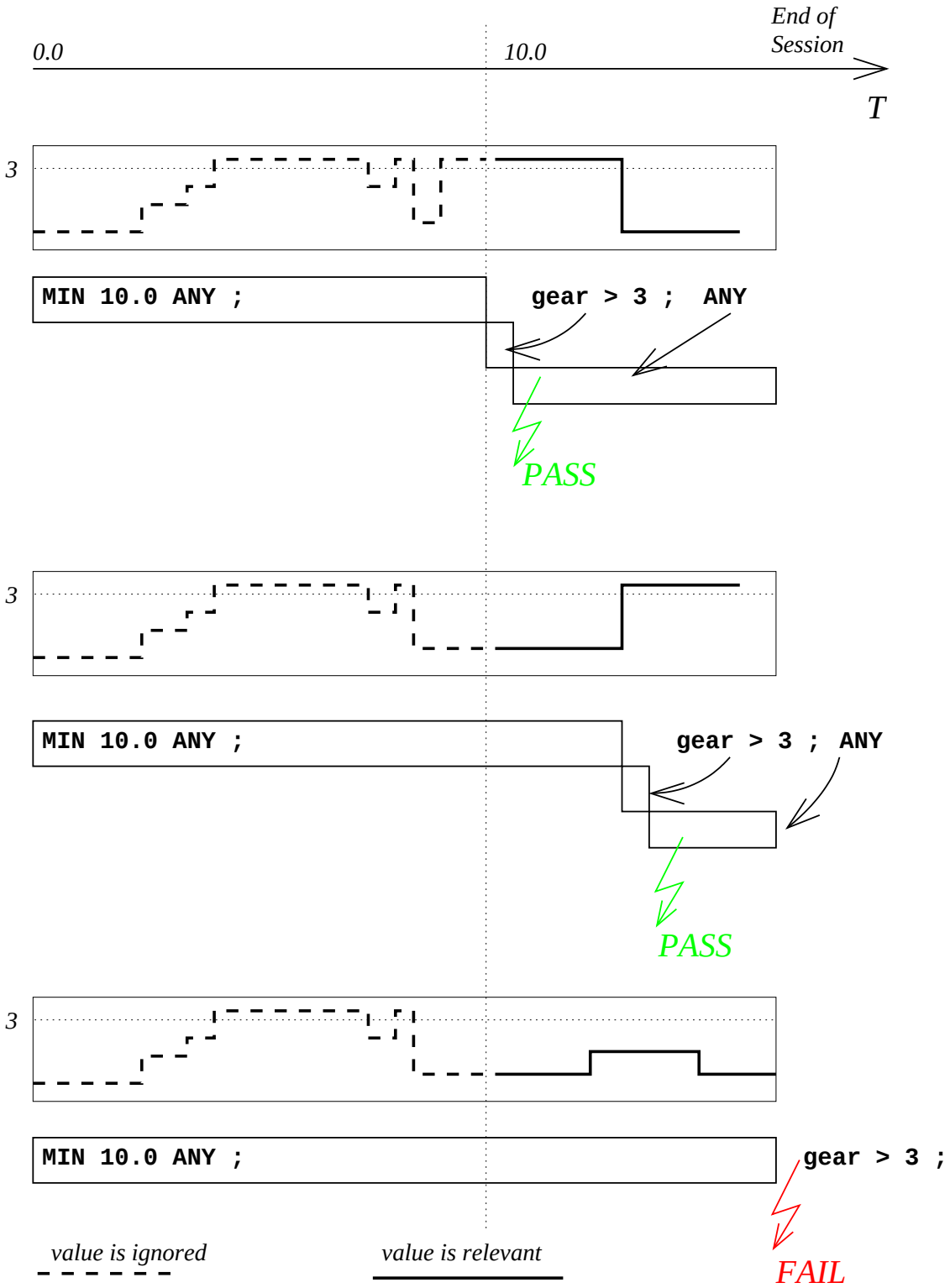
Or:

„The gear must be > 3 in some time interval, but the first 10.0 seconds do not count.“

This comes from the fact that at least the first 10.0 seconds are „consumed“ for the first sub-formula; the first segment of the segmentation „eats up“ the first 10.0 seconds (or more), and then, in the next segment, the condition must hold.

80 The price of this „direct denotation of trace semantics“ is obvious:

Figure 1 Possible Segmentations with the `MIN` formula from paragraph 79



You have to get used to it.

The benefits are not so obvious, but existent: you need not use variables or quantors, learn a block-oriented syntax, deal with lexical scopes and name clashes, etc.

81 The same formulation can of course be used concerning the *end* of the trace:

```
CASE  ANY ; shift_logic/gear > 3 ; MIN 10.0 ANY  EOF
```

means, that the gear has to be >3 somewhere more than 10.0 seconds before the simulation ends (because the last 10.0 seconds are consumed for the last sub-segment).

82 Another example, which now should be self-explaining :

```
LET  lowgear = shift_logic/gear <= 3 ;
CASE lowgear ;  MIN 3.0 ~ lowgear ; lowgear  EOF
```

requires that in some middle segment of the whole trace the gear must be > 3 for at least 3.0 seconds.

2.9 Specifying Maximally Allowed Durations of Sub-Traces

83 The `MAX` prefix works in analogy to the `MIN` prefix, as it defines the duration a subtrace of an allowed segmentation may maximally last.

As seen above, a `MIN` prefix of an instantanuos predicate p implies, that p is valid for minimally the given duration. The corresponding „dual“ statement is *not* valid.

The sentence „A `MAX` prefix of an instantanuos predicate p implies, that p is valid for maximally the given duration.“ *IS WRONG*, – cf. the different wordings of the section titles.

This may first be a suprise, – we hope it is understood after reading this section.

84 Let's consider the following example and look at the two possible cases of trace data depicted in figure 2 :

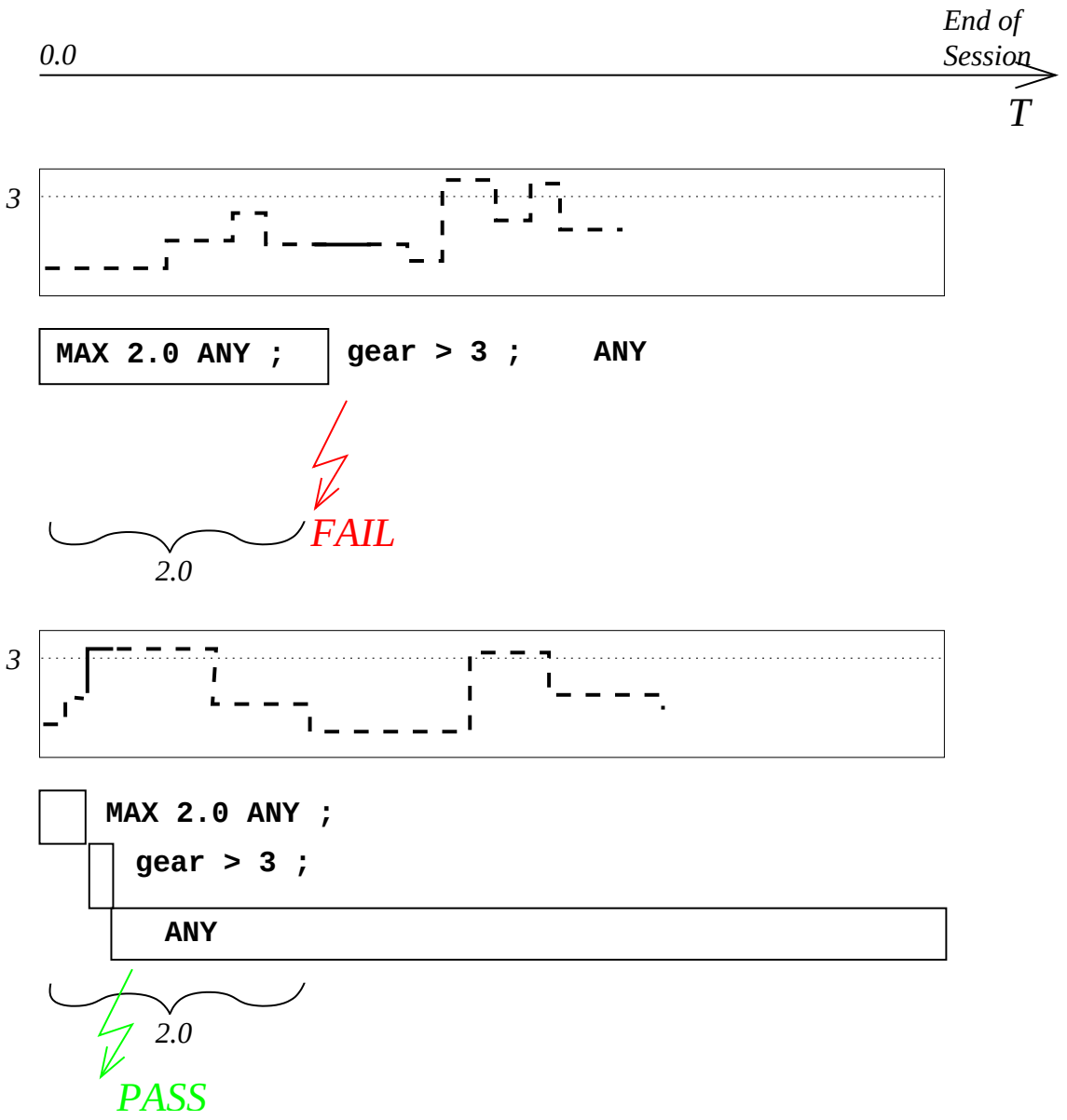
```
CASE MAX 2.0 ANY  ; shift_logic/gear > 3 ; ANY  EOF
```

Translated *verbatim* into the model language we get, that ...

„All traces are valid which can be divided into three segments: In the fist segment the system behaves arbitrary, but this segment may last maximally 2.0 seconds; then there is a segment in which the `gear` is larger than 3; the rest is arbitrary again“

This again can be translated to „normal language usage“ e.g. as ...

Figure 2 Trace Data with Possible and Impossible Segmentation for the `MAX` formula from paragraph 84



„All traces are valid, in which the `gear` is larger than 3 *not later than 2.0 seconds* after system start“.

This should be clear: As `ANY` allows the system to do anything, but `MAX 2.0 ANY` allows this only for maximally 2.0 seconds, and since this segment is followed by a segment where `gear` must be >2, then this instantanuos predicate must be true not later than the maximal extension of the first segment can reach.

85 This example also should clarify, that „`MAX` *d p*“ gives the time (*d*) a given instantanuos predicate (*p*) can maximally be „used“ or „considered“ for building a segment. It does not constrain the duration of the validity of the *p* itself: Otherwise a formula like `MAX` *d* `ANY` could never be fulfilled, since `ANY` is always valid.

86 A `MAX` prefixed *p* adjacent to an unconstrained `ANY` therefore never makes sense: As soon as the time allowed for the `MAX`-segment is exhausted, we put all the rest into the `ANY` segment and get a valid segmentation.

So the specification

```
LET  lowgear = shift_logic/gear <= 3 ;
CASE MAX 2.0 lowgear ; ANY ; MAX 2.0 lowgear  EOF
```

is totally equivalent to

```
CASE         lowgear ; ANY ;         lowgear  EOF
```

87 If one of the following examples uses `lowgear`, please assume that the line

```
LET  lowgear = shift_logic/gear <= 3 ;
```

is only left out for shortness.

88 Useful is the `MAX` construct by applying again a kind of „negation“ compared to „common sense“ usage of language :

```
CASE  lowgear  ;  MAX 3.0 ANY  ;  lowgear  EOF
```

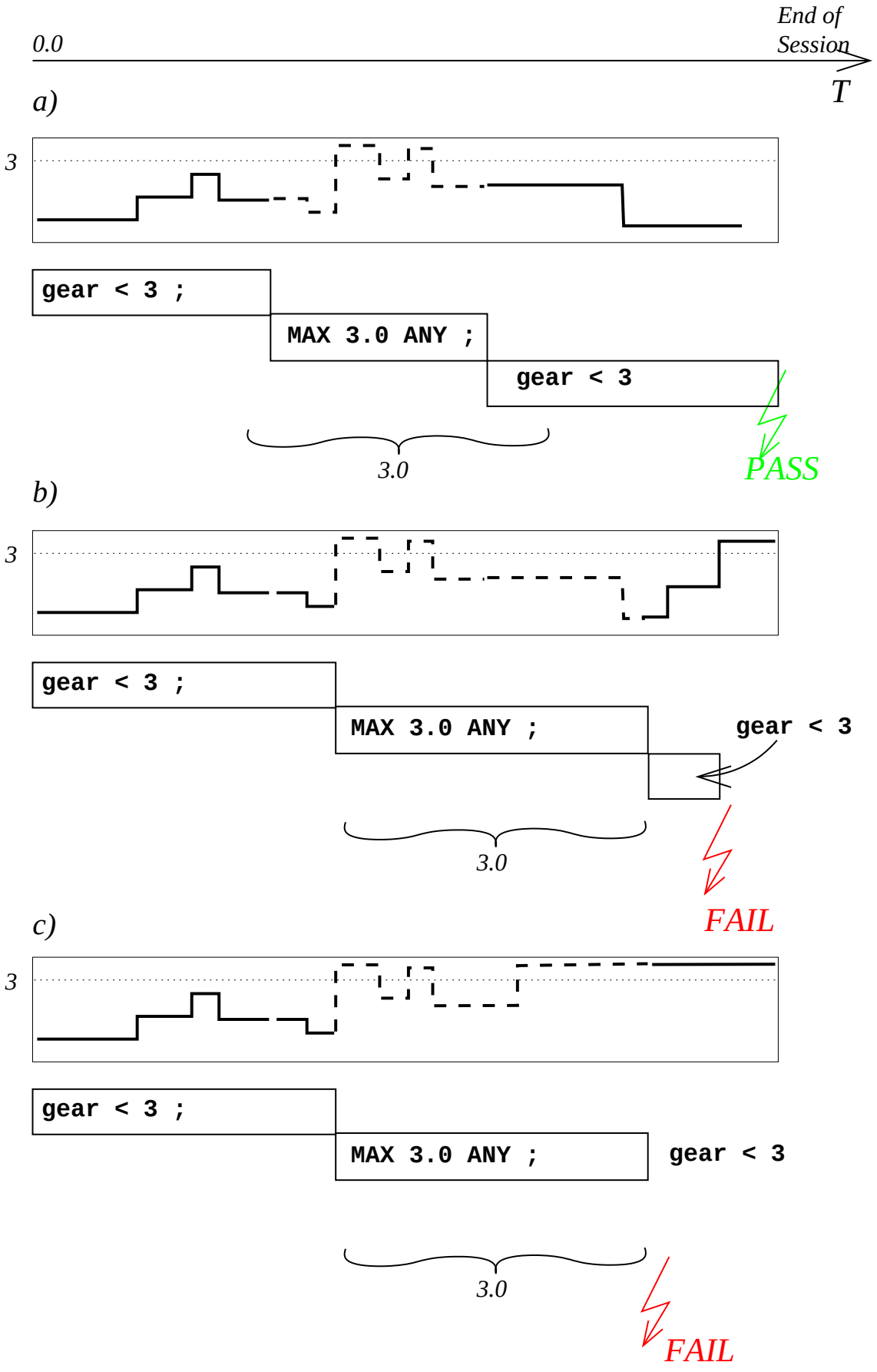
Since we claim for the gear being ≤3 in the first and last segment, only the middle segment would allow a gear larger than 3, because it does *not* impose this restriction.

Since this middle section can maximally have the length 3.0 seconds, we have expressed that the gear may never be larger than 3 for more than this duration.

Concerning the test data trace there are principally three different cases, depicted in figure 3:

In case a) there are *infinitively* many possible segmentations, which enclose the interval in which the gear is >3 in the middle segment, which corresponds to the "`ANY`" formula. Please note that this segment may extend beyond the critical points

Figure 3 Possible Segmentations with the `MAX` formula from paragraph 88



where `gear<=3` becomes false, and that it needs not „eat up“ the whole 3.0 seconds. The figure tries to indicate this graphically.

The opposite is true in cases a) and b): We have to shift the middle segment as far as possible into the future, e.g. let it start when the first segment *must* end, because its condition is no longer true. But even then it is not possible to find a valid segmentation, because the middle segment has to end after 3.0 seconds and the condition of the following segment is violated at some time instance.

89 The above specification *allows* the SSUT to change the gear to a value larger than three(3), but does not impose this behavior. Futhermore, the SSUT may switch to and fro a larger gear arbitrarily often, as long as the earliest and the latest time instance at which the gear is >3 *both fit into the* `ANY` *segment*.

90 If we want to express that the gear may only *once* cross this limit, we have to write a more restrictive specification (cf. 71 above) :

```
CASE  lowgear  ;  MAX 3.0  ~ lowgear  ;  lowgear   EOF
```

2.10 Comprehensing Sub-Traces

91 As seen above in in 72 the formula

```
CASE  lowgear  ;  ~ lowgear  ; ANY  ;  lowgear   EOF
```

expresses, that the trace data must start and end with a gear value ≤3, but that there is an intervall in between in which the gear *must* change at least once to be >3, but maybe more oftenly.

If we want to put a maximal duration constraint on this interval, the specification of which is itself a sequential composition of two predicates, we use the syntactic construct for defining *sub-traces*. This is done by enclosing the components of the subtrace in braces :

```
{ ... ; ... }
```

Now we can write

```
CASE  lowgear  ;  MAX 3.0 { ~ lowgear  ; ANY } ;  lowgear   EOF
```

The *whole subtrace* corresponding to the formula contained in {...} may *not last longer than* the given duration, that is: The interval in which the gear may take the value >3 must not be longer than three seconds, and starts with such a value.

W.r.t. the testdata the same three cases can occur as with the simple form "`MAX 3.0 ANY`" from paragraph 88, which are shown in figure 3. A difference is only case a), where the middle segment now has to begin as late as in the other cases.

92 The subtrace construct and its fully compositionality gives the MWATCH language their expressiveness. A typical example is

```
CASE  MAX 5.0  { MIN  2.0 lowgear  ; ~ lowgear } ; lowgear  EOF
```

The first two seconds have to be assigned to the first (sub-)segment in which `lowgear` holds. But the segment corresponding to the sub-formula must last at most 5.0 seconds. That means:

„The `gear` must be >3 exactly once, not earlier than 2.0 seconds after system start. 5.0 seconds after system start it must be low again."

93 Another pattern :

```
CASE  MAX 5.0  MIN  2.0 lowgear  ; ~ lowgear  ; ANY    EOF
```

. . . means . . .

„The *first time* the value of `gear` is >3 must happen not earlier than 2.0 and not later than 5.0 seconds after system start."

94 Another pattern :

```
CASE  MAX 5.0  { MIN  2.0 lowgear  ; ~ lowgear } ; ANY    EOF
```

. . . means . . .

„The *first time* the value of `gear` is >3 must happen not earlier than 2.0 and not later than 5.0 seconds after system start."

95 Another pattern :

```
CASE  MAX 5.0  { MIN  2.0 lowgear  ; ~ lowgear } ; ANY   EOF
```

2.11 Optional Segments and Optional Sub-Traces

96 As subtrace of a sequential MWATCH formula which is not preceded by a `MIN` constraint must nevertheless be existent. If such a subtrace – as in all our examples so far – contains an instantanuos predicate, there must be at least „one moment" in which this predicate is true. Technically this corresponds to one „sample point" of the simulation run, in which this predicate is true.

One could say that each predicate (occuring in a sequence) is implicitely prefixed by a „MIN ε“

$$\ldots \;;\; p_x \;;\; \ldots \quad \equiv \quad \ldots \;;\; \texttt{MIN}\ \varepsilon\ p_x \;;\; \ldots$$

This ε may be as small as necessary, but p_x has to be fullfilled *somewhere*[12].

If we want to specify a formula for a sub-segment which may occur, but does not need to, we have to use the keyword for declaring a subformula and its corresponding subtrace as *optional* :

```
OPT  s
```

97 With our example the way of operation of `OPT` can be seen if we use an instantanuos predicate definitively not fulfilled, like

```
CASE   ANY ; OPT  shift_logic/gear < -7 ; ANY  EOF
```

This specification *succeeds*, since the middle segment, preceded by `OPT`, needs not to be present at all in the actual trace data.

98 Let's recapitulate the patterns we found up to now for the different kinds of allowances we gave to the gear for being larger than 3.

```
CASE lowgear EOF
```
$\Longrightarrow$ not allowed at all.

```
CASE lowgear ; ANY ; lowgear EOF
```
$\Longrightarrow$ arbitrarily often.

```
CASE lowgear ; ~ lowgear ; lowgear EOF
```
$\Longrightarrow$ exactly once (cf. 71).

```
CASE lowgear ; ~ lowgear ; ANY ; lowgear EOF
```
$\Longrightarrow$ at least once (cf. 72).

Now we can also specify ...

```
CASE lowgear ; OPT ~ lowgear ; lowgear EOF
```
$\Longrightarrow$ at most once (i.e. once or never).

99 This can be combined with `MIN` and `MAX` constructs arbitrarily, i.e. our language is fully compositional.

An interesting case is

[12] This way of writing is for *convenience*: whenever the user writes an instantanuos predicate, he/she normally wants to express that this predicate is fulfilled. Semantically it is not quite orthogonal and requires the additional keyword "`OPT`", but otherwise the sepcifications would be quite unreadable, because everywhere a "`MIN` ε" would have to appear!

```
CASE lowgear ; OPT MIN 3.0 ~ lowgear ; lowgear EOF
```

Here the gear may stay low all the time, if the SSUT wishes to. But *if* it crosses the boundary, then it has to stay at least 3.0 seconds in a higher gear.

100 Do not mix this up with

```
CASE lowgear ; MIN 3.0 OPT ~ lowgear ; lowgear EOF
```

which is just identical with

```
CASE lowgear ; MIN 3.0     ~ lowgear ; lowgear EOF
```

The „option“ is no option: "`MIN`" d s requires that s lasts at least d timeunits. This forbids to leave out the subtrace `~ lowgear`, which would result to an „empty trace“. The length of an empty trace is 0.0, which is less than 3.0 and does not match the `MIN` constraint.

The example 99 before is the dual case: "`OPT`" s allows all traces allowed by s, plus the empty trace (length=0.0) *additionally*. So there was really an option[13].

2.12 Macros used as Abbreviations For Sequences

101 So the „macro definition“ feature of the MAWATCH compiler allows not only an abbreviated notation for predicates and expressions, as shown above in 75 *ff.*,but also for whole sequences.

The syntactic pattern is

LET i = { s } ;

We could write things like

```
LET lowgear    = shift_logic/gear <= 3 ;
LET lowdriving = { lowgear ; OPT MAX 3 ANY ; lowgear }  ;

CASE MAX 7 lowdriving ; MIN 15 ~ lowgear ; MIN 5 lowdriving  EOF
```

...specifying (a) that during a „`lowdriving`“ periode the SUT switches to a higher gear at most once for at most 3 time units, — and (b) that the SUT initially performs for at most 7 time units such a `lowdriving` behavior, than drives in a high gear for at least 15 time units, and finally becomes `lowdriving` again for at least 5 time units.

[13] There is still a **bug** with "`CASES { OPT` p_1 `; OPT` p_2 `} AND` p_3" !!

2.13 Repetition of Subtrace Specifications

REP { s }

102 Suppose we want to allow the SSUT to switch to a higher gear arbitrarily often, but only for a small duration, let's say 3 seconds.

This can be written like

```
CASE  REP {lowgear ; OPT MAX 3 ANY}  EOF
```

103 Now we want to describe the SSUT's behavior more precisely: The SSUT may change to a „higher" gear arbitrarily often, but in the middle of each run it must switch to a higher gear for a longer period.

```
LET lowgear    = shift_logic/gear <= 3 ;
LET lowdriving = REP {lowgear ; OPT MAX 3 ANY}  ;

CASE lowdriving ; MIN 5 ~ lowgear ; lowdriving  EOF
```

104 If we want to express, that the SSUT has to end the simulation run in a low gear, the formula above does not suffice, because the „lowdriving" segments may end in a higher gear. There are two possible variants:

```
LET lowgear    = shift_logic/gear <= 3 ;
LET lowdriving = REP {lowgear ; OPT MAX 3 ANY}  ;

CASE lowdriving ; MIN 5 ANY ; lowdriving ; lowgear EOF
```

105 ... or alternatively

```
LET lowgear    = shift_logic/gear <= 3 ;
LET lowdriving = REP {lowgear ; OPT MAX 3 ANY ; lowgear}  ;

CASE lowdriving ; MIN 5 ~ lowgear ; lowdriving  EOF
```

REPN n { s }

106 The `REPN` construct takes an integer *constant*, which gives the count of repetitions. By combining it with the `OPT` construct we can e.g. specify a maximal number of occurances of a predicate, as in

```
LET lowdriving = { REPN 3 {lowgear ; OPT MAX 3 ANY} ; lowgear }

CASE lowdriving ; MIN 5 ~ lowgear ; lowdriving  EOF
```

2.14 Disjunction of Sub-Formulae

107 Disjunction of two MATCH trace specification is done by the syntactic construct

$$\{\ \texttt{CASE}\ s_1\ \texttt{OR}\ \ldots\ \texttt{OR}\ s_n\ \}$$

108 Please not that if a disjunction (or conjunction, see below) appears at the top level of a specification, the leading keyword „`CASE`“ is omitted, so that the top level syntax appears as

$$\texttt{CASES}\ s_1\ \texttt{OR}\ \ldots\ \texttt{OR}\ s_n\ \texttt{EOF}$$

109 The meaning of *disjunction* can easily be taken over from the world of regular expressions: A specification which is the disjunction of two or more MATCH-formulae is fullfilled by the union set of the solutions of all alternatives.

Please notice the significant difference between the semantics (i.e. the set of fulfilling traces) of these both formulae:

```
CASES  lowgear  OR  speed > 120    EOF
CASE   lowgear  ||  speed > 120    EOF
```

The latter is fulfilled by all traces, in every instance of which the logical disjunction is true, i.e. one of the predicates „`lowgear`“ or „`speed > 120`“ is true.

The former but is fulfilled only by those traces which fulfill `lowgear` in their very first instance and *continue to fulfill this predicate* up to and including *the very last instance*, and those who do the same w.r.t. the predicate „`speed > 120`“

So the first specification implies that (at least) one of the instantantious predicates must evaluate constantly to `true` for the whole trace, while the latter allows the values of the p change arbitrarily often, as long as the boolean disjunction evaluates to `true`.

110 But the real meaning and importance should not become clear with this simple (and artificial) example, but in the complicated case, when combining *sequences* by the temporal-Or.

The following specification says that the SUT shall either be `lowdriving`, i.e. touching a higher gear only for a short duration, or otherwise be driving in higher gears for a longer period of time.

```
LET lowgear    = shift_logic/gear <= 3 ;
LET lowdriving = { REP {lowgear ; OPT MAX 3 ANY ; lowgear} }
LET phase      = { CASES lowdriving OR MIN 14 ~ lowgear }

CASE   phase ; phase ; phase EOF
```

. . . which expands to . . .

```
LET lowgear    = shift_logic/gear <= 3 ;
LET lowdriving = { REP {lowgear ; OPT MAX 3 ANY ; lowgear} }

CASE    { CASES lowdriving OR MIN 14 ~ lowgear } ;
        { CASES lowdriving OR MIN 14 ~ lowgear } ;
        { CASES lowdriving OR MIN 14 ~ lowgear }   EOF
```

. . . which expands to . . .

```
LET lowgear    = shift_logic/gear <= 3 ;

CASE    { CASES REP {lowgear ; OPT MAX 3 ANY ; lowgear}
          OR    MIN 14 ~ lowgear } ;
        { CASES REP {lowgear ; OPT MAX 3 ANY ; lowgear}
          OR    MIN 14 ~ lowgear } ;
        { CASES REP {lowgear ; OPT MAX 3 ANY ; lowgear}
          OR    MIN 14 ~ lowgear }                    EOF
```

111 In these situations a combination with `REP` and `REPN` seems senseful:

```
LET lowgear    = shift_logic/gear <= 3 ;
LET lowdriving = { REP {lowgear ; OPT MAX 3 ANY ; lowgear} }

CASE  REPN 3 { CASES lowdriving OR MIN 14 ~ lowgear }  EOF
```

. . . or similar . . .

```
CASE  REP { CASES lowdriving OR MIN 14 ~ lowgear }  EOF
```

112 With `REPN`/`REP` inline notation can be even more readable:

```
LET lowgear    = shift_logic/gear <= 3 ;

CASE  REPN 3 { CASES   REP {lowgear ; OPT MAX 3 ANY ; lowgear}
               OR      MIN 14 ~ lowgear }                    EOF
```

113 ⊕⊕⊕ As soon as temporal disjunction of *instantuos* predicates is embedded into a `REP` construct, it is not more powerful than a simple „logical“ or.

This comes from the fact the the `REP` construct must be applied to a *sequence* construct, and *not* to an instantanuos predicate or any disjunction of those.

```
CASE REP { CASES lowgear OR speed > 120 }   EOF
```

. . . means exactly the same as . . .

```
CASE             lowgear || speed > 120     EOF
```

114 ⊕⊕⊕ It is the outer „`REP`“, which levels the difference. The following formulae are *not* identical:

```
CASE REPN 3 {CASES lowgear OR speed > 120}   EOF
CASE REPN 3 {      lowgear || speed > 120}   EOF
```

The latter means exactly the same as . . .

```
CASE               lowgear || speed > 120     EOF
```

because, if this (logically or-ed) condition holds all the time, then there must be (an infinite number of) possible segmentations into three subtraces, in each of which the condition holds.

The first formula but imposes *additionally* a rather complicated restriction, requiring that the „responsibility of being continuosly true“ may maximally switch two times from one instantanuos predicate to the other, — in other words: That there is a possible semgentation into at most three segments, during each of which (at least) one of the two instantanuos predicates must be continuosly true.

115 ⊕⊕⊕ Consider further that also *sequencing* becomes equipotent to temporal OR and logical OR, as soon as `OPT` is used:

```
CASE  REP  { OPT lowgear  ;  OPT speed > 120 }  EOF
```

. . . really means the same as . . .

```
CASE  OPT REP  { CASE  lowgear OR speed > 120 }  EOF
```

. . . which — as seen above — is the same as[14] . . .

```
CASE  OPT              lowgear || speed > 120    EOF
```

[14] The following line is for illustration of semantics only and *not* valid MWATCH source, since a top-level `OPT` is rejected by the compiler.

2.15 Conjunction of Sub-Formulae

$$\{\ \texttt{CASES}\ s_1\ \texttt{AND}\ \ldots\ \texttt{AND}\ s_n\ \}$$

116 A specification which is the *conjuntion* of two or more MWATCH-formulae is fullfilled by the set intersection of the solutions of the sub-formulae.

In contrast to disjunction there is no corresponding notion of conjunction in the world of regular expressions.

117 The most simple application of the `AND` construct is just as a short hand notation for the repeated use of an instantanuos predicate:

```
LET gear    = shift_logic/gear  ;
LET speed   = "vehicle\nspeed" ;

CASES  gear=1 ; gear=2 ; gear=1 AND speed<30  EOF
```

. . . simply means the same as . . .

```
CASE  gear=1 && speed<30 ; gear=2 && speed<30 ; gear=1 && speed<30  EOF
```

118 A higher order of complexity but is reached by combining two (or more) sequences with the `AND` construct[15]:

```
CASES  gear=1 ; ANY ; gear=1  AND  ANY ; speed>=30 ; ANY  EOF
```

. . . matches all traces which (1) start and end with `gear=1`, and (2) reach `speed>=30` anywhere.

Please note that there is no specified relation between the „chop points“ (as denotated by the semicola) of the two sub-formulae.

2.16 MATLAB and simulink library functions

For the denotation of meaningful (instantanuos) predicates it is convenient, or even necessary, that the MWATCH language permits access to some built-in MATLAB and simulink functions.

[15] In deed the efficient implementation of the temporal conjunction is the main achievement realized by the MWATCH algorithm.

It is of central importance due to a role invisible to the user: All duration constraints on comprehensive sequences are realized by transforming them to a conjunction of the pure sequence (without timinig constraing) and an simple `ANY` carrying the duration constraint.

Table 4 List of supported MATLAB/simulink library functions

`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 their inverses
`diff`	"`(`" ⟨*arithExpr*⟩ "`)`" Derivation of given signal
`irgd`	"`(`" ⟨*arithExpr*⟩ "`)`" *Discrete* integration of given signal (Initial condition set to `0.0`.)
`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 ⟨*integerConst*⟩ gives the channel number of this newly created device, the value of which is used as the value of the expression.

These are listed in table 4.

The first group just performs „instantanuos“ calculations on some „instantanuos“ data.

The second group is more insteresting, since these library functions perform *time-related* processing of their input signals. In combination with the MATCH constructs for temporal trace specification they offer powerful specification possibilities, which will be discussed in section 2.17.

The third group onyl realizes some „technical“ interfaces to the rest of the MATLAB system, allowing to import test data from the „workspace“ or from a file.

2.17 Generating „Events“ from Discrete or Continuos Signals

119 For the following considerations we assume a semantics based on a *continuos* notion of time. A continuos signal $A(t)$ is defined as usual by the property that

$$\forall\ t_0 \quad \bullet \quad \text{l-lim}_{t \to t_0}\ A(t) \quad = \quad \text{r-lim}_{t \to t_0}\ A(t) \quad = \quad A(t_0)$$

Such a signal is depicted in figure 4 A.

A *discontinuos* signal $B(t)$ can be defined as a signal with a „jump“ or „gap“. At a certain time instance t_X a new value $B(t_X)$ is taken, but for all time instances before t_X (i.e. $t_X - \varepsilon$ for *aribtrary small* ε) the value is different.

$$\exists\ w > 0.0 \quad \bullet \quad \text{abs}\ (\ \text{l-lim}_{t \to t_X} B(t) \quad - \quad B(t_X)\ ) \quad \geq w$$

A *discrete* signal $C(t)$ can be defined as a discontinuos signal in which there is a non-empty interval of time between each two points of discontinuity, and which takes a *constant* value during these intervals:

$$\begin{array}{l} \forall\ t_1, t_2 \quad \bullet \quad t_1 < t_2\ \wedge\ C(t_1) \neq C(t_2) \quad \Longrightarrow \quad \exists\ t_3 \\ \qquad \bullet t_3 > t_1 \quad \wedge \quad \forall t\ |t_1 < t < t_3 \quad \bullet \quad C(t) = C(t_1) \end{array}$$

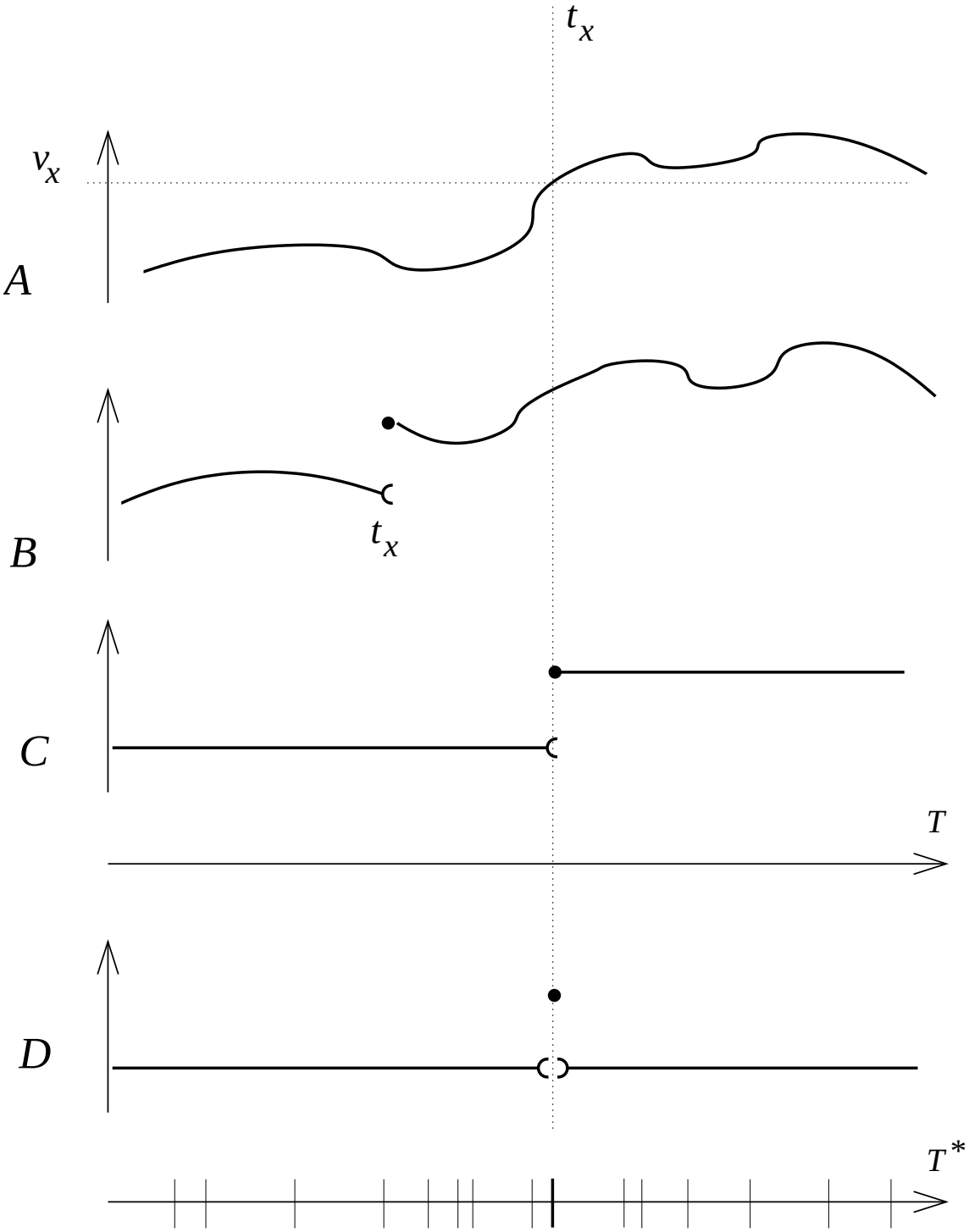
The signal `gear` in the preceding examples is such a „discrete“ signal.

120 In the simulink context a discrete function can be created by applying a relational operator to a continuos signal.

So figure 4 C shows the signal wich is derived from $A(t)$ by applying

$$\texttt{LET}\ \ C\ \ \texttt{=}\ \ A\ \ \texttt{>}\ \ v_X$$

Figure 4 Generating Discrete „Events“



121 All functions, continuos as well as discontinuos, are evaluated by the simulink execution model on different, unknown time instances. The only thing we propose is *Non-Zenoism*, i.e. that time advances and that for any time instance t (which lies in the inner interval of the total simulation time!) there is a simulation step which realizes a time instance $> t$. This step is preceeded by only a *finite* number of steps.

Therefore the set of evalution instances lying before t_X is finite, therefore there is always a *last* evaluation step before the point of discontinuity. Also there is always a *first* step, in which the current time is larger or equal to t_X.

Since a discrete signal (as defined above) does not change around its points of discontinuity, we can be sure that a derived signal like

$$\texttt{LET}\quad D\quad \texttt{=}\quad \texttt{memory}\quad \texttt{(}\ B\ \texttt{)}\quad \texttt{-}\quad B$$

differs from zero(0.0) only *in a single simulation step*, namely the first step after t_x, — see figure 4 C.

These kinds of signals, which take a specific value for a single step of the simulating machine, and which take another „neutral“ value for all steps immediately surrounding this time instance, can be called „events“.

Please note that by introducing this notion of event, i.e. by using the built-in simulink function `memory()`, we step over to a *totally different* world of semantics compared to all preceding discussions, indicated in figure 4 by using a different time axis T^*.

Now the *internal steps* of the simulink execution machine become part of the „visible“ semantics, since we know and take into consideration the fact that an „event-like“ signal takes its „non-neutral“ value only for one single *execution step*.

122 Fortunately this conceptual distinction does not affect the MATCH evaluation: MATCH treats events like any other condition and assumes a non-empty interval „around“ this simulink evalution step, during wich the condition (representing the event) does hold.

So we van define event-like signals as in ...

```
LET  gear      = shift_logic/gear ;
LET  noshift   = gear - memory(gear) = 0 ;
LET  shiftdown = gear - memory(gear) < 0 ;
LET  shiftup   = gear - memory(gear) > 0 ;

CASE noshift ; shiftup ; MIN 5.0 noshift ; shiftdown ; noshift EOF
```

... which matches all traces with exactly two gear shifts, one upshift followed by one downshift, both seperated by at least 5.0 time units.

123 Unfortunately the „discrete character“ of the `gear` signal, and the resultung „event character“ of the `shiftup` etc. is pure conceptual and not known to

MATCH.

So a sequence like

```
CASE noshift ; shiftup ; shiftup ; noshift EOF
```

does really mean the same as

```
CASE noshift ; shiftup ; noshift EOF
```

Both match to *arbitrary many* shift events, as long as they appear in adjacent simulation steps.

As seen by the MATCH semantics, all the „coming true“ of the event-like signal in adjacent simulink steps are just one single segment, during which the corresponding instantanuos predicate holds.

We know that there must be a non-empty idle interval between two gear shifts, — but as soon as you start the `sf_car` model with the wrong solver selected, there are many shifts in the first evaluation step.

So to specify that there must be more than one event, we have to seperate the predicates explicitely by their negation:

```
CASE noshift ; shiftup ; noshift ; shiftup ; noshift EOF
```

124 We can incorporate this „interval of constant value“ explicitly into our definition of the event:

```
LET  geardiff =  gear - memory(gear)  ;
LET  noshift  =  geardiff =  0 ;
LET  upshift  =  {geardiff =  1; MIN 0.1 noshift }
LET  downshift=  {geardiff = -1; MIN 0.1 noshift }
```

Now we can write

```
CASE noshift ; shiftup ; shiftup  EOF
```

thereby specifying two *distinct* shift events to occur.

2.18 Delegation of Non-Nondeterministic Calculations to simulink

125 Since MATCH operates in the world of non-determinism, it hardly makes any sense to „feed back“ signals generated by MATCH into the deterministic world of a running simulation.

But contrarily the basic expressions of many temporal predicates are time related or time dependent, but still deterministic. This is the case for all expressions which depend only on the signal flow produced by the SUT, not on the temporal context.

These predicates can be realized using the library functions mentionend above.

Some examples:

126

```
LET speed = "vehicle\nspeed" ;
LET gear         = shift_logic/gear ;
LET shiftup      = gear - memory(gear) > 0 ;
LET shiftupspeed = shold (speed, shiftup) ;
```

Here `shiftupspeed` memorizes the current speed at the time of the last upshift event.

127

```
LET speed = "vehicle\nspeed" ;
LET highaccel  = delay(speed, 2.0)*1.1 <  speed ;
LET highaccel1 = highaccel && ~ memory (highaccel) ;
```

Here `highaccel` is an boolean signal, indicating that the speed has increased in the last 2.0 timeunits by more than 10 percent.

`highaccel1` is an event type signal, indicating all time instances (which are simulink evaluation steps) at which `highaccel` switches from being false to being true.

2.19 Parametrized Macros and Local Macros

128 The denotation of more complex specifications requires appropriate means of abstraction. The macro mechanism of the MMATCH language, as introduced above in paragraph 75, offers such means by the mechanism of *parametrization*.

The following example specifies that each each trace starts with a gear ≤ 3, and that each upshift to the third gear has to be followed by a segment of at least 1.2 time units in which no shifts occur:

```
LET  gear      = shift_logic/gear ;
CASE REP { OPT gear < 3 ;
           OPT { MIN 1.2 gear = 3 ; gear >= 3 }
         } EOF
```

Now we can abstract this specification into a parametrized macro, and instantiate

it multiple times for specifying similar conditions for other gear levels:

```
LET  gear      = shift_logic/gear ;
LET  gear_up_pause (pgear, pdura) =
     { REP { gear < pgear  ;
             OPT { MIN pdura gear = pgear ; gear >= pgear }
           } }
CASES gear_up_pause (2, 2.2)
AND   gear_up_pause (3, 1.2) EOF
```

129 It may be useful to introduce local abbreviations inside the definition of a macro body, i.e. to define macros local to a macro definition, like in ...

```
LET  gear      = shift_logic/gear ;
LET  gear_up_pause (pgear, pdura) =
     { LET islower = gear < pgear  ;
       REP { islower ;
             OPT { MIN pdura gear = pgear ; ~ islower }
           } }
CASES gear_up_pause (2, 2.2)
AND   gear_up_pause (3, 1.2) EOF
```

Of course these macros could be again parametrized.