An Algorithm for the Real-Time Evaluation of Temporal Trace Specifications

vorgelegt von
Markus Lepper
aus Berlin

von der Fakultät IV — Elektrotechnik und Informatik —
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
— Dr.-Ing. —

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. B. Mahr
Berichter:	Prof. Dr. P. Pepper
Berichter:	Prof. Dr. S. Jähnichen

Tag der wissenschaftlichen Aussprache: 29. Juni 2004

Berlin 2004
D 83

Zusammenfassung

Diese Arbeit präsentiert und diskutiert einen Algorithmus, der das Verhalten eines beliebigen dynamischen Systems (= *System Under Test* = *SUT*) mit einer Spezifikation vergleicht, die eine Menge von erlaubten Verhaltensweisen beschreibt.

Der Algorithmus wird definiert durch eine rein funktionale mathematische Darstellung, und seine Vollständigkeit, Korrektheit, Terminierung und Konfluenz werden bewiesen.

Der Algorithmus arbeitet in Realzeit, insofern als er gleichzeitig mit dem SUT ausgeführt wird, und zu jedem Zeitpunkt ein Verdikt liefert, ob das Verhalten des SUT bis jetzt eine Erfüllung der Spezifikation noch erlaubt oder gar bereits impliziert.

Die Spezifikationen werden in einer eigenen, aber durchaus kanonischen Sprache erstellt, welche die bekannte Syntax und Semantik von regulären Ausdrücken erweitert um die konjunktive Verknüpfung und um die Bedingungen bezgl. minimaler und maximaler Dauer von Unterausdrücken. Ein Spezifikationsausdruck beschreibt eine Menge von erlaubten Verhaltensweisen unmittelbar, insofern als eine syntaktische Folge von Unterausdrücken direkt der zeitlichen Abfolge von Unterabschnitten des Verhaltens entspricht.

Die Ausdrucksfähigkeit der Spezifikationssprache entspricht einer temporalen Intervallogik zuzüglich Dauernanforderungen als *first class residents*, aber ohne Negation. Die Grundlage ihrer Semantik und der Arbeitsweise des Algorithmus ist die Arithmetik von Intervallen über $\mathbb{R}$.

Der Algorithmus bedarf, um auf ein beliebiges System angewandt zu werden, der Implementierung einer jeweils entsprechenden Adaptiven Schicht. Die Anforderungen an diese werden in der Arbeit spezifiziert.

Der Algorithmus ist Kernbestandteil des im industriellen Kontext entwickelten Werkzeugs *MWATCH*, welches als Bibliotheksbaustein für die MATLAB/simulink-Umgebung implementiert ist, und für die Auswertung von Testdaten eingesetzt werden soll. Darüber hinaus enthält das Werkzeug eine Instanz der Adaptiven Schicht, welche im Haupttext erläutert wird, und realisiert eine programmierbare und zwei graphische Benutzerschnittstellen, welche in den als Anhang beigefügten Handbüchern beschrieben werden.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Subject of this text

The following text presents, discusses and proves the correctness of an algorithm which compares the behaviour of a arbitrarily defined dynamic system (= *system under test* = *SUT*) with a specification (= *SpecUT*) which describes a set of permitted behaviours.

The specification is given as an expression from a corresponding *specification language*. The expressiveness of this language is that of a temporal interval logic over a dense domain, including duration specifications as first class residents, and excluding negation. The basis of its semantics, and of the operation of the algorithm, is the arithmetic on intervals from $\mathbb{R}$.

The algorithm works in *real-time*: while monitoring the growing prefix of the known behaviour of the SUT, it frequently generates *verdicts* indicating whether the complete behaviour of the SUT will finally fulfill or violate the specification, or whether this is currently still inconclusive.

The real-time instant, when the algorithm's execution starts, serves as reference point for the semantics of the specification. An ending time instant of the SUT's behaviour may or may not be given, i.e. the algorithm can monitor finite or infinite real-time intervals of execution.

Any kind of SUT can be monitored by the algorithm, if an *adaptive layer* is provided which transforms the observation data into the required input format.

Currently both, the algorithm and an instance of this adaptive layer, are implemented in the *MW*ATCH tool. This tool is realized as a so-called "function block" in the MATLAB/simulink environment [12][13]. Its development has been financially supported by DaimlerChrysler/FT3/SM, where simulink models are used for model-based development, and *MW*ATCH shall be used in course of automated test evaluation.

Since the algorithm described herein is the central part of any such tool implementation, it is called *kernel algorithm* in the following. Currently (November 2003) the author is applying for a European patent on the kernel algorithm.

1.2 Structure of this text

The following text is structured as follows :

- Chapter 2 describes the conditions an environment has to fulfill to make the kernel algorithm applicable, and the rules for calling its interfaces.
- Chapter 3 defines the language of the specifications treatable by the algorithm.
- Chapter 4 gives an informal description of the algorithm's operation and of discusses the major design decisions.
- Chapter 5 presents the algorithm as a collection of mathematical functions.
- Chapter 6 proves the correctness, completeness, confluence and termination of the algorithm.
- Chapter 7 shows the differences of the solution contained herein to other approaches.

Since the formulæ constituting the algorithm are frequently referred to by the proofs in chapter 6, it has been considered more convenient for the reader to present them in an outmost compact way in their own dedicated chapter 5, and not to mix them with explanations which all are gathered into chapter 4.

A survey of the globally used notations and abbreviations, and the technical manual of the *MW*ATCH tool and a tutorial on writing specifications, both requested by DaimlerChrysler/FT3/SM, are included as appendices.

Chapter 2

Context of the Algorithm's Operation

2.1 Real-Time Input Data and Adaptive Layer

The algorithm is applied to an SUT by combining the executions of both during a certain interval of real-time. This interval is called *session interval* (or *session*) in the following. Each session interval has a certain known time instant as its starting point.

During a session the information which represents the behaviour of the SUT flows from the SUT to an adaptive layer, and from the adaptive layer to the kernel algorithm, see figure 2.1.

The observable behaviour of an SUT is constituted by a collection of functions from the session interval into arbitrary ranges. These functions are called *SUT functions* in the following.[1]

The kernel algorithm takes as input data (1) an abstract syntax representing the specification, (2) a few additional configuration parameters, and (3) a real-time data stream, representing the SUT's behaviour.

Data of kind (1) and (2) are passed to the algorithm once at set-up time, i.e. shortly before or exactly at the time instant when the session starts.

The third kind of input data, the real-time data stream, consists of a discrete representation of a finite indexed collection of functions from the session interval into the set of *Boolean* values. These functions are called *observation functions* (or simply *observations*) in the following.

The collection of all observation functions used in a certain session is called *trace* in the following. All collections of these functions restricted to one common cohesive, non-zero interval of real-time are called *sub-traces* of the trace, and are also considered to be traces.

The objects used for indexing the observation functions of a given trace are called *atomic predicates* in the following.

[1] In the context of test evaluation and of industrial tools, these functions are often called *PTOs*, i.e. "Points of Test and Observation". This wording is also used in the tutorial in appendix C.

It is the task of the adaptive layer to continuously derive the required observation functions from the SUT functions.

While there are no further assumptions on the structure of the SUT functions, it is an essential requirement for the operation of the kernel algorithm that all observation functions are of *finite variability*, cf. section 4.7.7 on page 30.

This means that in each observation function the points of discontinuity are separated by a distance not smaller than a certain, positive distance δ_{SEP}. This property is also known as *non*-ZENO*ism*. This requirement is only existentially qualified: The value δ_{SEP} must exist theoretically, but it need not be known in advance, nor does its value influence the semantics of the algorithm.

The transmission of the current values of the observation functions from the adaptive layer to the kernel algorithm is realized by a *discrete* representation: Whenever the former detects at a certain time instant a discontinuity of one or more observation functions, it calls an interface procedure (*iNotify()*, see section 4.1 below). This call is parameterized (1) with a time stamp value, identifying the current time instant, and (2) a collection of Boolean values indexed by all atomic predicates. These values are those which the corresponding observation functions will take immediately after this point of discontinuity.

Due to the finite variability of all observation functions it is guaranteed that these newly taken values will stay stable for some non-zero duration.

2.2 Deriving Observation Functions from SUT Functions

Basically there are two kinds of SUT functions from which observations can be derived, namely *analog signals* and *event streams*.

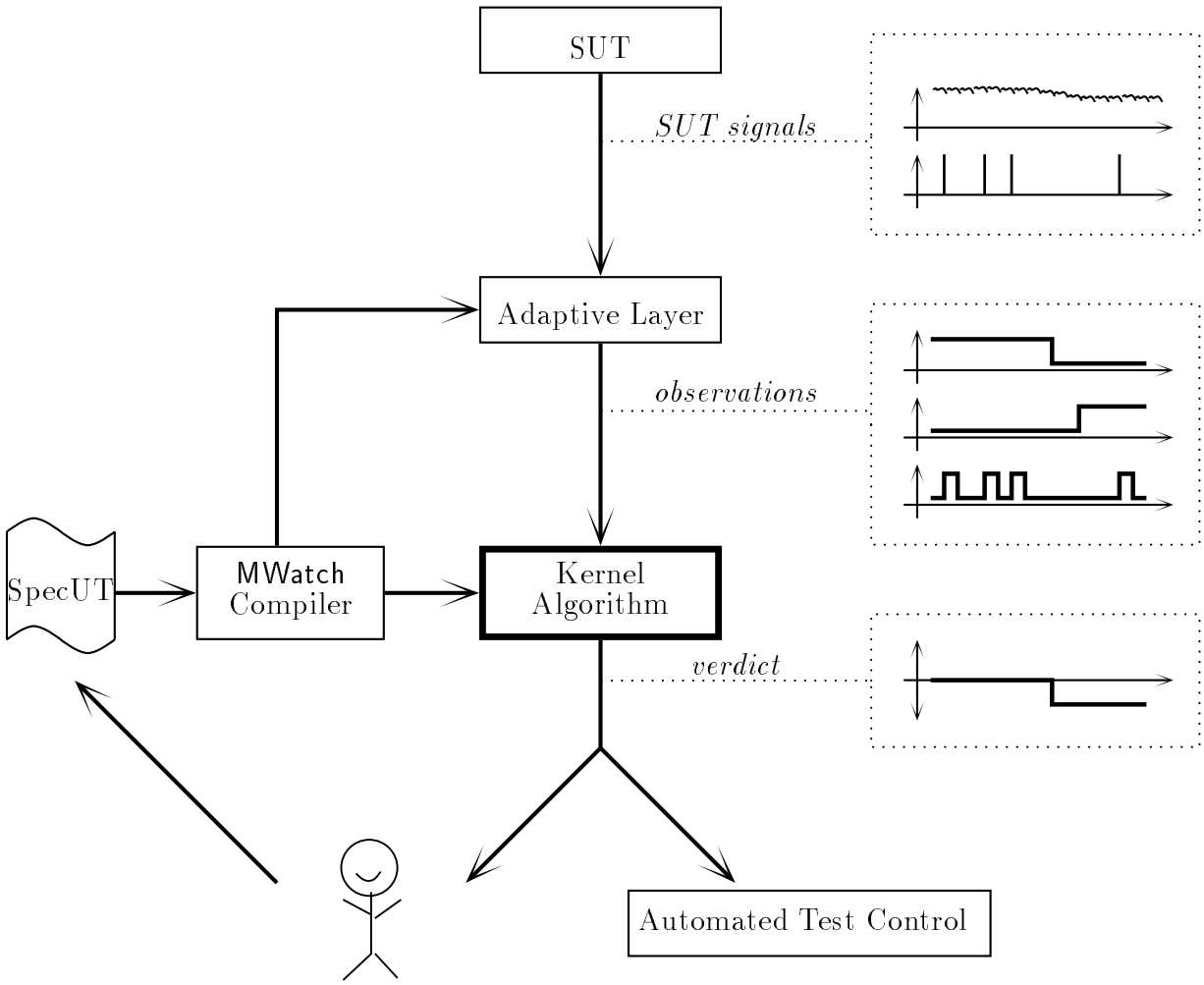
Analog signals may be either produced by analog/digital-converter hardware (ADCs), when observing a real physical system, or by a digital simulation model which delivers representations of analog values in discrete steps using e.g. an equation solver.

In both cases the technical representation is discrete, namely a sequence of pairs of time instants and range values, while — just contrarily — the intended semantics are those of a total function from a dense domain into a dense scalar range. According to SHANNON's theoreme [18], the mapping between representation and semantics is unique, if a certain maximal bandwidth of the intended dense function is assumed.

From SUT functions of this kind the Boolean valued observation functions can be derived by applying arithmetic comparison operators either to one or two SUT functions directly, or after feeding them through arbitrarily defined signal processing networks.

In most cases it is advisable to use only the comparison operators $<$ and $\leq$, and to avoid the usage of $=$, i.e. the exact equality. With the former, the bandwidth of the resulting observation function is almost totally determined by the intended

Figure 2.1 Flow of Data: Specification, SUT Signals, Observations and Verdict



semantics. But when using the exact equality, the result depends highly on the kind of discrete representation chosen by the implementation layers of the SUT, e.g. the ADC drivers or the kind of solving algorithm currently selected by the simulation framework, — things which should be abstracted from when dealing with the semantics of a model.

Event streams can be regarded as functions from time to Boolean, which take the value "true" only at single, isolated points of the dense real-time domain. These SUT functions can be translated into valid observation functions by replacing each such spike by the positive edge of a pulse of arbitrary non-zero duration.

The specification term passed to the kernel algorithm must be written accordingly, so that it waits only for the positive edges of the observation function for detecting an SUT event, and uses negative edges only for distinguishing between subsequent events in the same SUT function.

2.3 Configuring the Adaptive Layer in the *MW*ATCH Tool

For the kernel algorithm, the adaptive layer and the translations from SUT functions into observation functions are totally invisible.

But from the user's point of view these translations correspond to the definitions of the atomic predicates, which are an integral part of their specification.

This point of view is supported by the current implementation of the *MW*ATCH tool: its input language integrates the means for defining all the different processing steps mentioned above into one single front-end representation.

This corresponds to a layered structure of the syntax definition of the tool's input language:

- As basic elements there are *path expressions*, each of which addresses a certain outlet of a certain simulink function block contained in the SUT. So these expressions directly correspond to the SUT functions.
- From these elements — together with denotations of numeric constants — arithmetic and signal processing expressions can be constructed.
- On these expressions comparison operators can be applied, yielding Boolean valued functions.
- These can further be combined using logical operators, finally yielding the observation functions.
- At the top level, the behavioural specification is constructed by combining the observation functions using the temporal operators from the kernel specification language (see chapter 3 below).

The compiler part of the *MW*ATCH tool translates a specification text into a sequence of MATLAB commands, the execution of which inserts a so-called "masked sub-system" into the simulink model representing the SUT.

This sub-system includes the instance of a predefined function block, which encapsulates the implementation of the kernel algorithm, and a signal processing network realizing the adaptive layer according to the user's specification.

In the compilation process the different layers of the specification are separated:

- The top level temporal specification is compiled into the input format of the kernel algorithm's implementation. This format includes only a skeleton, consisting of the temporal combinators of the kernel specification language, and of atomic predicates which uniquely identify observation functions.
  All expressions below the level of the temporal combinators are treated as implicit definitions of observation functions and replaced by the corresponding atomic predicate.
- The arithmetic expressions, signal processing commands, comparison operations and logical combinations contained in these extracted definitions of observation functions, are translated into MATLAB code which set up the signal processing network of the sub-system accordingly.
- Each path expression addressing a function block outlet, i.e. an SUT function, is translated into MATLAB commands which attach a so-called "goto block" to this outlet. Thereby the signal of this SUT function is fed into the processing network of the created sub-system.

Consider as an example the following fragment of a specification in the tool's specification language (the temporal combinators `MIN`, `MAX` and `;` will be explained in chapter 3):

```
...;  MAX 5 abs(vehicle_speed - 3.2*throttle) >= MYCONST ;
      MIN 0.25 MAX 1.25 engine/rpm - delay(engine/rpm, 2.7) < 35 ; ...
```

This fragment will be translated into two different groups of definitions, notated symbolically like ...

p_4 = ...
p_5 = `abs(vehicle_speed - 3.2*throttle) >= MYCONST ;`
p_6 = `engine/rpm - delay(engine/rpm, 2.7) < 35`
p_7 = ...

`...;` `MAX 5` p_5 `; MIN 0.25 MAX 1.25` p_6 `; ...`

Figure 2.2 shows the optical appearance of an *MW*ATCH function block and some "goto" blocks, inserted into one of the standard demonstration models contained in the MATLAB/simulink distribution, — figure 2.3 shows a corresponding signal processing network hidden behind the mask of the "masked sub-system".

For a complete syntax for defining the signal processing processing network of the adaptive layer, please refer to the Technical Manual included as appendix B.

Figure 2.2 The *MW*ATCH Tool Applied to a MATLAB/simulink Standard Example

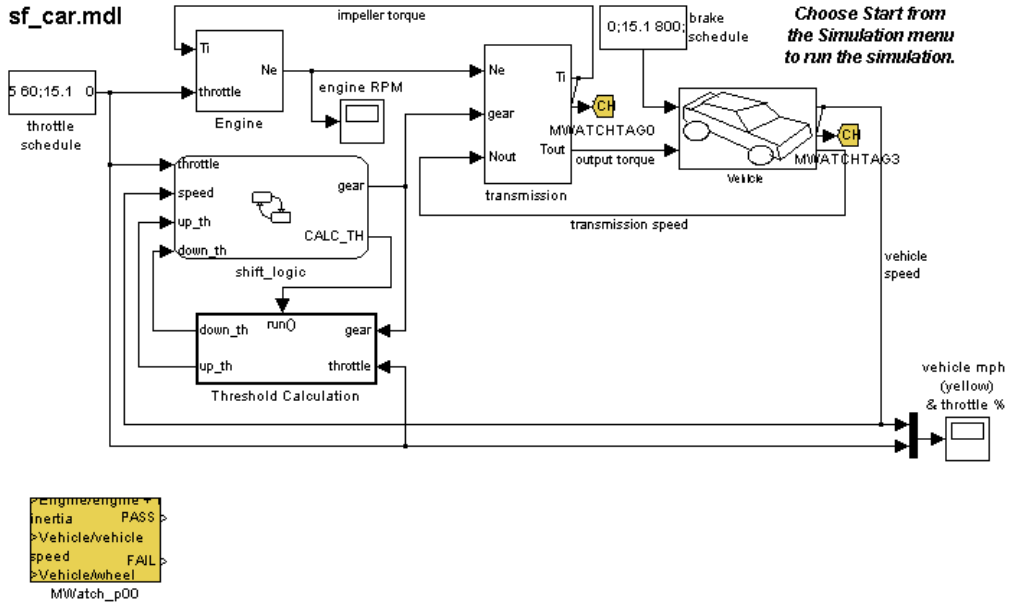
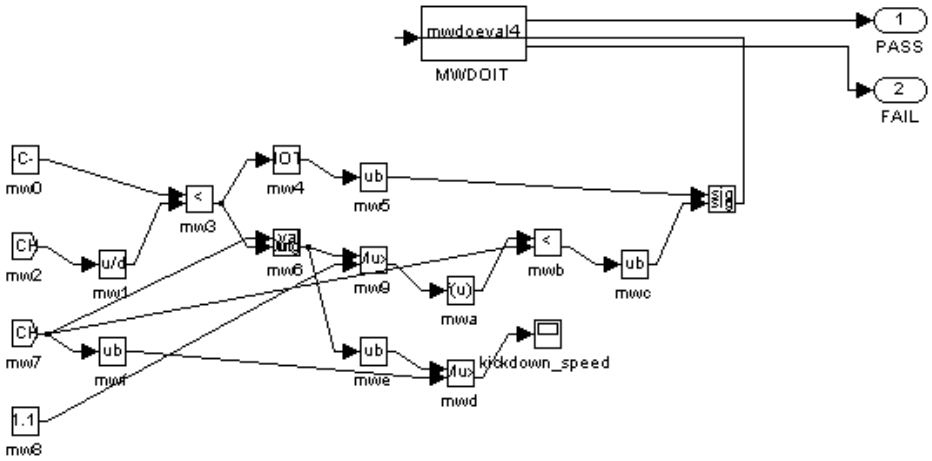


Figure 2.3 A Signal Processing Network Realizing an Adaptive Layer



Chapter 3

The Temporal Specification Language

3.1 Syntax

The specifications which can be processed by the kernel algorithm are all finite terms which are produced by the following recursive rule:

$$
\begin{array}{lcl}
S & ::= & p_k \;\mid\; \texttt{ANY} \\
 & \mid & \texttt{MIN}\, d\, S \;\mid\; \texttt{MAX}\, d\, S \;\mid\; S_1 \,\underline{;}\, S_2 \\
 & \mid & \texttt{OR}\,\{S_1\,,\ldots,\,S_m\} \;\mid\; \texttt{AND}\,\{S_1\,,\ldots,\,S_n\} \\
 & \mid & \texttt{REP}\, S \;\mid\; \texttt{OPT}\, S \\
\\
p_k & ::= & p_1 \mid p_2 \mid p_3 \mid \ldots
\end{array}
$$

3.2 Informal Semantics

The semantics of each specification term corresponds to the set of traces which *fulfill* this specification[1].

Applying the kernel algorithm to a certain specification and to a trace which is known to be *complete* (which means that the test session which produces the trace is finished) can yield two different results, which are called *final verdicts*: If the trace fulfills the specification, the algorithm will yield the verdict `pass`, otherwise it will yield `failed`.

Since the algorithm works in real-time, it is frequently supplied with a steadily growing prefix of the trace, representing the "already known" prefix of the behaviour of the SUT. In the case that this prefix is not yet complete, i.e. the session is known to be continued, the verdict generated by the algorithm is called *early verdict value*, and has a slightly different meaning:

[1] The notion of *trace* has been defined above in section 2.1. A trace is a collection of observation functions, defined on one common, cohesive interval of real-time, and represents the behaviour of an SUT during a (sub-interval of a) session.

- `pass` means that the trace will fulfill the specification in any case, i.e. each possible completion of the prefix trace will yield a trace which fulfills the specification.
- `fail` means that there exists no single completion of the prefix which will fulfill the specification.
- `inconc` (read "inconclusive") means that the algorithm is not yet able to decide between these two cases. This case is discussed in more detail in section 6.5.3.

The verdicts `pass` and `fail` are commonly called *conclusive verdicts*.

The semantics of the syntactic constructs in terms of the fulfillment relation are defined as follows:

- An atomic predicates p_k identifies an observation function, as described above in section 2.1. Used as a specification term, it is fulfilled by all those traces in which the value of this observation function is continuously `true`.
- The specification `ANY` is fulfilled by any trace.
- The constructs `MIN` $d\,S$ and `MAX` $d\,S$ represent *duration requirements*. The value of d has to be some positive, non-zero numeric value. This kind of specification is fulfilled by all those traces which fulfill S and additionally have a duration which is larger or equal / less or equal to d.
- The *chop construct*[2] $S_1 \underline{;} S_2$ is fulfilled by all traces which can be divided at some inner time instant into two adjacent sub-traces, the first of which fulfills S_1 and the second fulfills S_2.
- A specification like `AND` $\{S_2, S_1\}$ is called *conjunctive specification* and is fulfilled by all traces which fulfill S_1 and S_2.
- A specification like `OR` $\{S_1, S_2\}$ is called *disjunctive specification* and is fulfilled by all traces which fulfill S_1, or which fulfill S_2, or which fulfill both.
- The specification `REP` S denotates the disjunction of arbitrarily many, non-zero repetitions of S combined by the chop operator $\underline{;}$. Its semantics would be identical with those of the infinite specification term[3] `OR` $\{(S), (S \underline{;} S), (S \underline{;} S \underline{;} S), (S \underline{;} S \underline{;} S \underline{;} S), \ldots\}$.
  It is fulfilled by all traces which fulfill one or more of these chop expressions.
- The specification `OPT` S makes sense only as an argument of the chop constructor $\underline{;}$: it denotates an *optional specification* which *may* be considered part of the specification, but which does not need to be fulfilled if this is not required by the context.
  Indeed it is just a convenient front-end shortcut notation for a disjunction construct. E.g. the specification $p_1 \underline{;}$ `OPT` $p_2 \underline{;} p_3$ is equivalent to `OR` $\{(p_1 \underline{;} p_2 \underline{;} p_3), (p_1 \underline{;} p_3)\}$.

For some instructive examples of specifications written in this language S and for some illustrating diagrams of the timing of early verdicts please refer to the tutorial which is included as appendix C.

[2] The naming "chop operator" has first been used in the definition of the duration calculus, cf. [3] and chapter 7, which is on related work.

[3] This infinite term itself is not a member of the front-end specification language, which is restricted to finite terms derivable from S.

3.3 Formalized Semantics

The formal definition of the semantics of the specification language is based on the definitions of data types and auxiliary functions in table 3.1. In the following formulæ the μ operator, which selects the single element contained in a given set, the treatment of functions as relations, and the mapping operator $(\!|\ldots|\!)$ are borrowed from the Z notation [19].

The basic data types are the set of *Boolean* values, the set $\mathbb{T}$ for representing time, and the set $\mathbb{D}$ for representing durations, i.e. positive or zero-valued distances between time instants.

The set P_+ contains all atomic predicates indicating an observation function, while the set P additionally includes p_0, an internally defined auxiliary atomic predicate which corresponds to an observation function which for each time instant always delivers `true`. This object is used later in the implementation to treat the specification `ANY` in a uniform way like the atomic predicates.

lb and ub deliver the sets of lower and upper bounds for a given set of time instants, and glb and lub are functions which deliver the (always uniquely defined) greatest lower bound and least upper bound.

Using these functions the set of all non-empty traces $\mathcal{R}_+$ is defined as the set of all functions from time $\mathbb{T}$ and the atomic predicates P to the *Boolean* values, which are restricted to a certain non-empty interval, and which are in this interval total w.r.t $\mathbb{T}$ and to P.

The set of traces $\mathcal{R}$ additionally includes the empty trace, which is written simply as an empty relation $\{\}$. It is used solely for modeling the semantics of the `OPT` operator.

The function $\mathsf{conc} : \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}$ is needed for modeling the chop operator: It takes two traces and concatenates them. In case that one of the traces is empty, the concatenation result is just the other trace. In case that both traces are non-empty, (1) the domain of the second trace is shifted by composing it with an appropriate transposition function, so that it starts exactly when the first trace ends, and then (2) both functions are combined by superposition.

Note that the definition of $\mathcal{R}$ does not specify whether the domain intervals are open, closed or half-open. Therefore the domains of both functions are disjoint after the transposition of the second trace, except possibly for the point $\mathsf{lub}(r_1)$, where possibly both functions are defined or undefined.[4]

The function $\mathsf{combine}$ operates on collections of traces, and delivers the concatenations of all combinations between the elements of both sets.

For each non-terminal N from a syntax defining grammar the expression $\mathcal{L}\,N$ shall denotate the corresponding language, i.e. the set of all derivable finite sentences.

Then the semantics of the grammar S as defined in section 3.1 can be formally

[4] Because of the finite variability of all observation functions, cf. page 4 above, both kinds of conflicts can be resolved by choosing one of only two canonical alternatives. Strictly spoken, conc delivers a set of one or two resulting traces. For the sake of readability, this idiosyncratic fact has not been formalized in table 3.2 and 3.1.

Table 3.1 Data Types and Auxiliary Functions for Defining the Semantics of Specification Expressions

$$
\begin{array}{lcl}
\mathit{Boolean} & = & \{\texttt{false}, \texttt{true}\} \\
\mathbb{T} & = & \mathbb{R}_{\geq 0.0} \\
\mathbb{D} & = & \mathbb{R}_{\geq 0.0} \\
\\
P_+ & = & \{p_1, p_2, \ldots, p_{maxp}\} \\
P & = & P_+ \cup \{p_0\} \\
\\
\mathsf{lb}(T : \textbf{set of } \mathbb{T}) : \textbf{set of } \mathbb{T} & = & \{t : \mathbb{T} \mid (\neg\ \exists\, t_2 \in T \bullet t_2 < t)\} \\
\mathsf{ub}(T : \textbf{set of } \mathbb{T}) : \textbf{set of } \mathbb{T} & = & \{t : \mathbb{T} \mid (\neg\ \exists\, t_2 \in T \bullet t_2 > t)\} \\
\mathsf{glb}(T : \textbf{set of } \mathbb{T}) : \mathbb{T} & = & \mu\{t \in \mathsf{lb}\ T \mid (\neg\ \exists\, t_2 \in \mathsf{lb}\ T \bullet t_2 > t)\} \\
\mathsf{lub}(T : \textbf{set of } \mathbb{T}) : \mathbb{T} & = & \mu\{t \in \mathsf{ub}\ T \mid (\neg\ \exists\, t_2 \in \mathsf{ub}\ T \bullet t_2 < t)\} \\
\\
\mathcal{R}_+ & = & \{r : \mathbb{T} \rightarrow P \rightarrow \mathit{Boolean} \\
 & & \mid \mathsf{glb}\ (\mathsf{dom}\, r) < \mathsf{lub}\,(\mathsf{dom}\, r) \\
 & & \wedge\ \forall\, t \in \mathbb{T} \mid \mathsf{glb}\ (\mathsf{dom}\, r) \leq t < \mathsf{lub}\,(\mathsf{dom}\, r) \\
 & & \qquad \Longrightarrow\ t \in \mathsf{dom}\, r \ \wedge\ P = \mathsf{dom}(r\ t)\} \\
\mathcal{R} & = & \mathcal{R}_+ \cup \{\{\}\} \\
\\
\mathsf{conc}(r_1 : \mathcal{R}, r_2 : \mathcal{R}) : \mathcal{R} & = & \left\{ \begin{array}{l} \textbf{if } r_2 = \{\} \textbf{ then } \ r_1 \\ \textbf{if } r_1 = \{\} \textbf{ then } \ r_2 \\ \textbf{otherwise} \\ \quad r_1 \oplus (r_2 \circ (\lambda\, t \bullet t - \mathsf{glb}\ (\mathsf{dom}\, r_2) + \mathsf{lub}\ (\mathsf{dom}\, r_1))) \end{array} \right. \\
\\
\mathsf{combine}\,(w_1 : \textbf{set of } \mathcal{R}, w_2 : \textbf{set of } \mathcal{R}) : \textbf{set of } \mathcal{R} & & \\
 & = & \lambda\ x_1, x_2 \bullet \mathsf{conc}\,(x_1, x_2)\ (\!|w_1 \times w_2|\!)
\end{array}
$$

defined by a function $[[\_]]^L : \mathcal{L}\,S \rightarrow \mathbb{P}\,\mathcal{R}$, which maps each specification expression to the set of those traces which fulfill this specification.

This function is defined in table 3.2. The informal description of the semantics from section 3.2 can be used as a legend: The wording "the trace $r : \mathcal{R}$ fulfills the specification $s : \mathcal{L}\,S$" used above is totally equivalent to the statement "$r \in [[s]]^L$".

Table 3.2 Formal Semantics of the Specification Language S

$[[\_]]^L$	:	$\mathcal{L}\,S \;\rightarrow\; \textbf{set of}\ \mathcal{R}$
$[[p_k]]^L$	=	$\{r \in \mathcal{R}_+ \mid \forall\, t \in \mathsf{dom}\, r \bullet r(t)(p_k) \;=\; \mathtt{true}\}$
$[[\mathtt{ANY}]]^L$	=	$\mathcal{R}_+$
$[[\mathtt{MIN}\, d\ \mathtt{ANY}]]^L$	=	$\{r : \mathcal{R} \mid \mathsf{glb}\ (\mathsf{dom}\, r) - \mathsf{lub}\ (\mathsf{dom}\, r) \geq d\}$
$[[\mathtt{MAX}\, d\ \mathtt{ANY}]]^L$	=	$\{r : \mathcal{R} \mid \mathsf{glb}\ (\mathsf{dom}\, r) - \mathsf{lub}\ (\mathsf{dom}\, r) \leq d\}$
$[[\mathtt{MIN}\, d\ \ s]]^L$	=	$[[\mathtt{MIN}\, d\ \mathtt{ANY}]]^L \;\cap\; [[s]]^L$
$[[\mathtt{MAX}\, d\ \ s]]^L$	=	$[[\mathtt{MAX}\, d\ \mathtt{ANY}]]^L \;\cap\; [[s]]^L$
$[[s_1\ \underline{;}\ s_2]]^L$	=	$\mathsf{combine}\,([[s_1]]^L, [[s_2]]^L)$
$[[\mathtt{AND}\ \{s_1, \ldots, s_n\}\ ]]^L$	=	$[[s_1]]^L \cap \ldots \cap [[s_n]]^L$
$[[\mathtt{OR}\ \{s_1, \ldots, s_n\}\ ]]^L$	=	$[[s_1]]^L \cup \ldots \cup [[s_n]]^L$
$[[\mathtt{REP}\ s]]^L$	=	$[[s]]^L \;\cup\; [[s\ \ \underline{;}\ \ \mathtt{REP}\, s\ ]]^L$
$[[\mathtt{OPT}\ s]]^L$	=	$[[s]]^L \;\cup\; \{\{\}\}$

Chapter 4

Informal Description of the Kernel Algorithm

4.1 Interfaces and Usage

The kernel algorithm is constituted by the definitions of three *interface functions* the implementations of which must be called by an implementation of the adaptive layer, as sketched out above in section 2.1 on page 4.

The algorithm operates by applying transformations to a *state*.

In the definitions of the kernel algorithm in chapter 5 this state is modeled as a value of type *GState*.

Since the algorithm is defined as a collection of pure functions, this state must be explicitely threaded by the adaptive layer through all calls of the algorithm's interface functions. The adaptive layer must handle this value transparently, which means to use the same, unmodified state value for each function call which has been returned by the preceding function call.

The three interface functions have to be used as follows:

- Before the start of a test session the interface function *iInit*() is called for creating an initial state for the algorithm, according to the specification term passed as an argument. This specification term is called *SpecUT* in the following.
  Additionally the maximal duration of the test session can be given to *iInit*() by the parameter *maxSessionDuration*, which allows an earlier detection of an early `pass` verdict. If this value is not known, the special value ∞ is given as parameter value.
- For the time instant when the test session is started, and for each subsequent time instant when at least one observation function changes its current value, the interface function *iNotify*() must be called. Its arguments, beside the transparently preserved state object, are (1) a time stamp value identifying the current time instant and (2) the collection of the newly taken values of all observation functions, indexed by the corresponding atomic predicate.

Table 4.1 Syntax of the Kernel Algorithm's Input and Internal Language S'

$$
\begin{array}{lcl}
S' & = & s\_or \\
 & & \\
s\_and & = & \mathtt{AND}_k \; \{ \, s\_or^+ \, \} \\
s\_or & = & \mathtt{OR} \;\; \{ \, s\_alt^+ \, \} \\
s\_alt & = & s\_seq \;\mid\; \mathtt{REPst}\,(s\_seq \mid s\_and) \;\mid\; s\_base \\
s\_seq & = & s\_step \; (\underline{;} \; s\_step)^+ \\
s\_step & = & s\_opt \;\mid\; s\_base \\
s\_opt & = & \mathtt{OPT} \; (s\_base \mid s\_seq) \\
s\_base & = & s\_and \\
 & \mid & {}^{i,a}p_k \;\; \textbf{where} \; i \leq a \\
 & \mid & \triangle_{\triangle} \;\mid\; \triangle_{\mathcal{L}\, s\_seq}
\end{array}
$$

This interface function returns an early verdict value, which is one of the values $\{\mathtt{pass}, \mathtt{fail}, \mathtt{inconc}\}$, indicating the fulfillment relation between the specification term and the prefix of the SUT's trace, as far as it is currently known, and all its possible continuations (cf. the description of verdict values above in section 3.2 on page 9).
As soon as a conclusive verdict is returned, the algorithm's behaviour is not longer defined.

- If a session has ended and no conclusive early verdict has been returned, a final verdict is calculated by calling the interface function *iFinalize*(). The final verdict is always conclusive.
  Note that for the time instant representing the end of the test session the interface function *iNofify*() may *not* be called. This is because the algorithm makes extrapolations concerning the subsequent future time interval, which would not be correct in this case, cf. section 4.7.7 below.

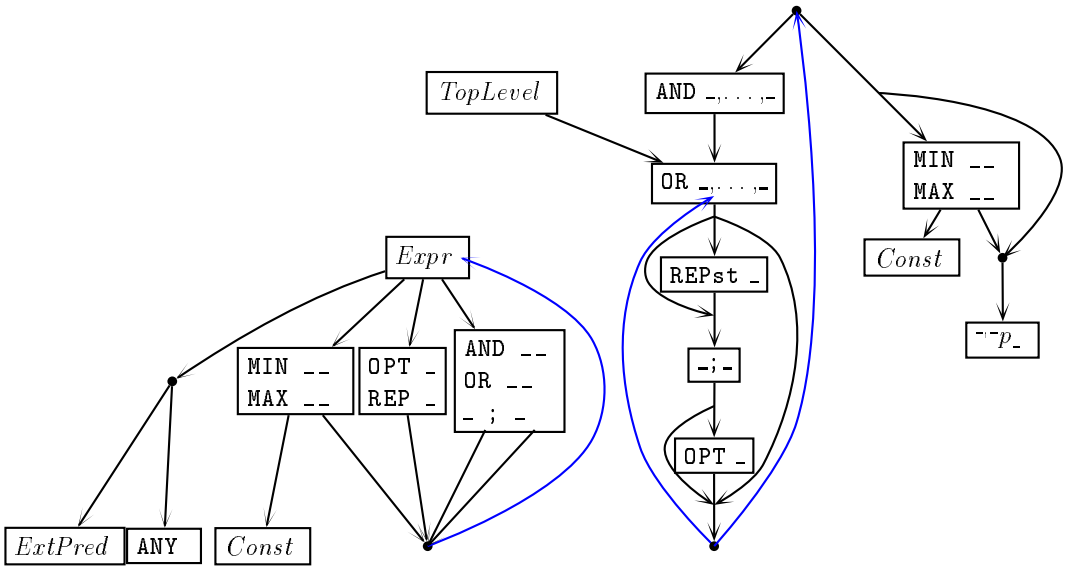
4.2 Normalization of Specification Terms

The specification expression SpecUT passed to the interface function *iInit*() must be given in a *normalized form*, which must be a sentence of the language S' as defined in table 4.1. It must be a sentence of a subset of $\mathcal{L}\,S'$, because it must not contain one of the terminal symbols $\triangle_{\triangle}$ and $\triangle_{\mathcal{L}\,S'}$, which are reserved for the internal use of the algorithm.

While the possibility of arbitrary combination ("freeness") of constructors in the front-end language S is an important issue for the author of the specification, the language S' restricts these combinations. This allows a straight-forward definition of the real-time algorithm, and thus increases its efficiency and readability. The different structure of expressions in both languages is depicted in figure 4.1.

The semantics of all those constructs in S' which are known in S are carried without any changes from the semantics defined for S in table 3.2.

Figure 4.1 The Grammars of S and S' Represented Graphically



The semantics of the new constructs in S' are defined as ...

$$
\begin{array}{lcl}
[[\texttt{REPst}\, s]]^L & = & [[\texttt{REP}\, s]]^L \cup \{\{\}\} \\
[[{}^{i,a}p_k]]^L & = & [[\texttt{MIN}\, i\, \texttt{MAX}\, a\, p_k]]^L
\end{array}
$$

The suffix `st` has been chosen as a mnemonic for the "star" operator.

It is always possible to define a transformation from $\mathcal{L}\,S$ to $\mathcal{L}\,S'$, which is total and semantic preserving. In a practical implementation such a transformation could and should include further simplifications and optimizations, as it is the case with the MWATCH tool. An exhaustive and formalized discussion of these transformations is neither challenging nor in the scope of this work.

In any case transformations are required concerning ...

- the re-writing of expressions using `OPT`, `REP` and `;` , possibly in combination with duration requirements,
- the extraction of duration requirements,
- and the canonicalization of conjunctions and disjunctions,

A first group of transformations simplifies the use of `OPT` and `REP` expressions: The `REP/REPst` constructors only make sense when applied (directly or via an `AND` expression) to chop expressions. Just contrarily, each `OPT` expression must be an argument to a chop expression. In all other cases the expression is re-written according to the defined semantics, possibly eliminating these operators and modifying duration requirements.

Furthermore, each expression of type `ANY` is replaced by the reserved atomic predicate p_0, which can be seen as representing an observation function which always takes the value `true`. This allows the algorithm to treat all atomic expressions in a uniform way.

In this section the term *atomic expression* refers to all expressions in S which are either atomic predicates p_k in S or S', or of the form `ANY` in S, and the term *complex expression* refers to all expressions in S which are constructed by the application of `AND`, `OR`, `REP` or $\underline{;}$.

Then the main transformation replaces duration requirements on complex expressions by a conjunction of this expression (or possibly of some sub-expression, in case of an `AND`-expression) with a duration requirement on an atomic expression.

For instance, ...

$\texttt{MIN}\, d\, (p_8 \underline{;} p_9 \underline{;} \ldots)$	is replaced by	$\texttt{AND}\{\, (\texttt{MIN}\, d\, \texttt{ANY}), (p_8 \underline{;} p_9 \underline{;} \ldots)\,\}$
$\texttt{MAX}\, d\, \texttt{REP}\ (p_8 \underline{;} p_9 \underline{;} \ldots)$	is replaced by	$\texttt{AND}\{\, (\texttt{MAX}\, d\, \texttt{ANY}), \texttt{REP}(p_8 \underline{;} p_9 \underline{;} \ldots)\,)\,\}$
$\texttt{MIN}\, d\, \texttt{AND}\ \{(p_7), (p_8 \underline{;} p_9 \underline{;} \ldots)\,\}$	is replaced by	$\texttt{AND}\{\, (\texttt{MIN}\, d\, p_7), (p_8 \underline{;} p_9 \underline{;} \ldots)\,\}$
$\texttt{MIN}\, d\, \texttt{OR}\ \ \{\alpha_1, \alpha_2, \ldots\}$	is replaced by	$\texttt{AND}\{\, \texttt{MIN}\, d\, \texttt{ANY}, \texttt{OR}\{\alpha_1, \alpha_2, \ldots\}\,\}$

Since after this transformation step only atomic expressions are subject to duration requirements, the constructors `MIN` and `MAX` are eliminated from the language: each expression $\texttt{MIN}\, i\, \texttt{MAX}\, a\, p_k$ is rewritten to ${}^{i,a}p_k$. In case that there is no minimal(/maximal) duration requirement imposed on p_k, the value of $i(/a)$ is set to 0.0 $(/\infty)$. Additionally the transformation ensures that $i \leq a$, which is of central importance for the efficiency of the algorithm, cf. formula (6.26).

A next group of rewriting steps assures that (1) the top level expression is an `OR` term, that (2) each `AND` term only contains `OR` expressions, and that (3) all `OR` terms except for the top level one are immediately contained in an `AND` term.

The purpose of this group of transformations is the following:

During its operation, the algorithm continuously calculates all possible mappings between the SUT's trace and the SpecUT (="partial interpretations", as defined in section 4.7 below). A specification like $\alpha \underline{;} {}^{\cdot\cdot}p_k$ can correspond to multiple different ones of these mappings, if the observation function oscillates between `true` and `false`, while α is continuously fulfilled by the trace data, — as it is schematically depicted in figure 4.2 on page 26.

For the sake of simplicity of the algorithm's definition, this kind of non-determinism shall be handled by the same means as the explicit non-determinism caused by `OR` expressions contained in the original specification. This is achieved by wrapping the top level specification expression and all those arguments of `AND` expressions which are not yet `OR` expressions into a unary `OR` expression, and all `OR` expressions which are not argument of an `AND` expression into a unary `AND` expression.

Because both these constructors occur in $\mathcal{L}\, S'$ only in this combination, the `AND` expressions will be referred to by the wording "`AND/OR` expression " in the following text.

4.3 Node Objects and Evaluation Steps

The central component of the above-mentioned state the kernel algorithm works on, is a collection of *node objects* (or simply *nodes* in the following).

In the definitions of the kernel algorithm in chapter 5 this collection is modeled the attribute *GState.nodes*.

It is the basic idea of the algorithm, that in each time instant of the session interval the current state of this node collection reflects the fact whether the SUT's behaviour up to this instant permits or even implies the complete trace to fulfill the SpecUT.

Node objects carry two kinds of attributes: local attributes of different scalar domains (time instants, durations, and atomic predicates represented as integer numbers) and attributes which refer to other node objects (see section 5.3).

During the execution of the algorithm operation, new nodes are created and added to this collection, attribute values of existing nodes are altered, and nodes are removed from the collection.

The state of this collection of nodes is changed in reaction to (1) the change of the current value of observation functions, as made known to the algorithm by calls to the *iNotify*() interface function, and (2) to the expiration of *timer requests*, which are maintained internally and set up according to the duration requirements contained in the specification term.

The algorithm consists of a collection of function definitions, each of which belongs to one of three groups:

- The interface functions (as described above in section 4.1 and defined in section 5.4) and the top level *scheduling functions* (see section 5.5),
- functions defining the transformations applied to the node collection in each *evaluation step* (sections 5.7 to 5.9),
- functions which *analyze* the current state of the node collection to derive a verdict value (see formulæ (5.11) and (5.12)).

Central part of the algorithm is the definition of an *evaluation step*. Only when an evaluation step is executed the state of the node collection possibly changes.

The *scheduling functions* directly implement the interface functions described at the beginning of this chapter (see section 4.1 above). When executed, they initiate the execution of one or more evaluation steps in an appropriate order, and finally call an *analyzing function* for calculating a verdict from the resulting state of the node collection and returning this to the caller.

The execution of an evaluation step is always related to a certain time instant, and all evaluation steps have to be executed in non-decreasing order of the corresponding time instants.

An evaluation step *must* be executed for each time instant at which (1) the value of at least one observation function changes, or (2) at least one timer request expires, or (3) several of these kinds of events happen simultaneously. These time instants are called *critical* in the following.

If at a critical time instant the values of more than one observation function change, this fact has to be signaled by the adaptive layer to the algorithm *completely* in one single call to *iNotify*(), attributed with all these new values.

Multiple calls to *iNotify*() for the same time instant are permitted, but the argument containing the current values of the observation functions must be identical for all these calls.

The first execution of an evaluation step at a critical time instant definitely changes the state of the node collection. Subsequent executions for the same time instant will not have any further effect. Executions of an evaluation step for a time instant which is not critical do not have any effect on the state of the node collection either.

4.4 Internal and External Scheduling of Timer Requests

As mentioned above, the expirations of timer requests are handled internally to the algorithm. Since they always cause an evaluation step, they possibly result in a conclusive verdict value.

The kernel algorithm offers to the adaptive layer two ways of dealing with timer requests:

iNotify() additionally returns the time stamp of the timer request which is the earliest to expire. In case that this time instant is reached before the change of an observation function has caused an evaluation step anyway, the adaptive layer may call *iNotify*(), — of course with the currently valid set of observation values, which is unchanged w.r.t. that of the preceding call — and thus trigger the execution of an evaluation step for inquiring the possibly conclusive verdict caused by the timer expiration.

But this behaviour of the adaptive layer is not necessary for the correct operation of the algorithm[1]. This is because the scheduling function implementing *iNotify*() always considers whether there have been critical time instants between the time instants of its last and of its current execution, i.e. time instants which are critical only due to timer expirations and not due to changes of observation functions. For all these time instants one evaluation step each is executed in the correct sequential order, before finally the evaluation step for the current time instant is executed.

The same rules of executing evaluation steps apply when *iFinalize*() is called for the time instant corresponding to the end of the test session for all critical time instants later than the time instant of the last call of *iNotify*.

Note that this external triggering of evaluation steps has in no concern any influence on the semantics, e.g. w.r.t. the accuracy of duration measurement. The internal scheduling of the algorithm is always executed independently. The only effect of the external scheduling is that the caller of the interface functions might get a conclusive verdict earlier. Due to the idem-potence of the evaluation step the adaptive layer might even call *iNotify*() in arbitrary random intervals, as long as the rules listed above in sections 4.1 and 4.4 are respected.

[1] Indeed, in certain technological contexts this behaviour would not be adequate.

Table 4.2 Syntax of Linear Specifications S''

$$
\begin{array}{lcl}
S'' & = & s\_seq \\
 & & \\
s\_seq & = & s\_base\ (\underline{;}\ s\_base)^* \\
s\_base & = & {}^{i,a}p_k \ \mid\ s\_and \\
s\_and & = & \mathtt{AND}_k\ \{\ s\_seq^+\ \}
\end{array}
$$

4.5 Internal Structure of an Evaluation Step

Each evaluation step consists of two different, strictly separated phases:

In the first phase, called *positive phase*, new nodes are created and added to the collection.

In a second phase, called *negative phase*, existing nodes are removed from the collection.

Additionally, in both phases the state of already existing nodes can be subject to some minor and local alterations.

The timer requests caused by a `MIN` expression in the specification are called *time-in requests*, those caused by a `MAX` expression are called *time-out requests*.

In the positive phase all reactions to the expiration of time-in requests and to the becoming-true of observation functions are performed. Each single event of both kinds can lead to the creation of none, one or finitely many node objects.

In the negative phase all reactions to the becoming-false of the observation functions are performed, followed by all reactions on the expiration of a time-out request.

Each single event of both kinds can lead to the removal of arbitrary many of the currently existing node objects, which are always of finite number.

4.6 Linear Specifications and Interpretations

The notion *expanded specification* denotates the expression which is derived from SpecUT by replacing all `REPst` expressions and all chop expressions containing `OPT` expressions by the corresponding `OR` construct (see section 3.2 on page 10 above, and the description of the implementation in section 4.7.5 on page 27 below).

Due to the definition of the `REPst` constructor, the resulting term may include `OR` expressions which are not finite.[2]

The notion *linear specification* denotates a specification which is derived from the expanded specification by replacing each `OR` construct by one of its alternatives. A linear specification is always a finite term.

[2] Therefore, strictly spoken, an expanded specification is possibly not an expression from the front-end specification language as defined in the previous chapter, and which includes only finite terms.

The language of linear specifications which results from these transformation can be described by the syntax S'' given in table 4.2. Obviously, the chop operator $\underline{;}$ is not longer required in $\mathcal{L}\,S''$, and *s_seq* could have been defined simply as s_base^{+}. This means that a linear specification is just a *sequence* in the mathematical sense of elements from $\mathcal{L}\,s\_base$. This is the view which will be taken in the rest of this text.[3] Such a sequence is called *chop sequence*, and its elements are referred to as its *(specification) particles*.

Every specification is semantically equivalent to the disjunction of all the different linear specifications which are derivable from it.

The algorithm works by monitoring the SUT's trace data w.r.t. all linear specifications derivable from SpecUT.

Let s be a linear specification of k particles. An *interpretation* i of given trace D w.r.t. s is a sequence of $k+1$ time instants $\langle t_1, \ldots, t_{k+1}\rangle$, for which it holds that $t_1 = \mathsf{glb\,dom}\,D$ and $t_{k+1} = \mathsf{lub\,dom}\,D$, and that i cuts the trace D into k non-empty sub-traces $\langle g_1, \ldots, g_k\rangle$, such that every nth sub-trace fulfills the nth particle of s. In the context of an interpretation, each such sub-trace is called a *segment*. The nth segment and the nth sub-expressions are mutually called *corresponding to*. In the same context, the time instant value t_m is called the *start time* of the segment g_m, and t_{m+1} its *end time*.

Let for the rest of this text v_k be the observation function indicated by the atomic predicate k, i.e. the observation function corresponding to all specification expressions $^{--}p_k$.

If the nth particle is of form $^{i,a}p_k$, the existence of the interpretation is equivalent to the fact that during the whole nth segment v_k is continuously `true`, and that the length of this segment is larger or equal to i and less or equal to a.

If the nth particle is of form $\texttt{AND}\,\{\alpha_1, \ldots, \alpha_m\}$ the existence of the interpretation means recursively that there exist different interpretations of the corresponding segment, at least one w.r.t. each $\alpha \in \{\alpha_1, \ldots, \alpha_m\}$.

The collection of interpretations of a given trace w.r.t. one certain linear specification is in most cases of infinite cardinality : If $\langle t_0, t_1, t_2\rangle$ is an interpretation of D w.r.t. the outmost simple specification $p_1 \underline{;} p_2$, and if v_1 and v_2 are simultaneously true ("overlap") during a non-zero interval around t_1, then each time instant from this interval can be substituted for t_1, yielding infinitely many interpretations.

The existence of at least one interpretation of a given trace D w.r.t. a linear specification e is equivalent to the fact that D fulfills e, in the sense of the denotational semantics presented in section 3.2.

Therefore, a given trace fulfills a given specification, iff there exists at least one interpretation of this trace w.r.t. at least one of the linear specifications derivable from this specification.

[3] For sake of readability, instances of chop sequences may nevertheless be notated using the "$\underline{;}$" symbol as delimiter.

4.7 Operation of the Kernel Algorithm

4.7.1 Notational Conventions

The text in this section explains the operation of the kernel algorithm and the design decisions taken therein caused by semantical considerations.

Since it frequently refers to the formulæ of the next chapter, which constitute the algorithm by mathematical means, interjections like "(cf. formula (5.47))" would frequently occur in the text.

For sake of a more fluent readability, these interjections are in most cases replaced by the short-cut notation "$^{(5.47)}$". In an electronic version of this text these references would be replaced by hyper-links.

A juxtaposition of these references means a sequence of function calls, so $^{(5.18)(5.19)(5.21)}$ reads as "cf. function (5.18), which calls (5.19), which in turn calls (5.21)".

If no call chain but a mere enumeration is meant, the notation is $^{(5.8)+(5.39)}$.

The same shortcut notation is used in chapter 6.

4.7.2 Partial Interpretations and the Semantics of Node Objects

During the execution of the algorithm, a *partial interpretation* is an interpretation of a prefix of the SUT's trace data w.r.t. a certain prefix of a linear specification derivable from SpecUT.

The operation principle of the algorithm is to model *all* partial interpretations by node objects.

Each node object belongs to one of three classes, called `Prime` nodes, `ATst` nodes and `ASol` nodes , — see formula (5.6) and the graphical notation in figure 5.1. `Prime` nodes and `ASol` nodes are commonly referred to as *LNodes*, and they represent partial interpretations which end with a segment corresponding to an atomic predicate or to a conjunctive expression, resp. `ATst` nodes represent conjunctive expressions for which further interpretations are still possible.

At each time instant t a node object is in a certain *state*. The definitions of the possible states are specific to the different node classes. The state of a certain node may alter during the execution of an evaluation step.

Some of the states defined for *LNodes* are special and called *valid states*. An *LNode* which is in such a valid state at a time instant t is simply called a (currently) *valid node*.[4]

[4] In a strict sense, this wording can only be used if t does not correspond to an evaluation step which alters the node's state. In this case, all function calls in the positive and negative phase would have to be considered in detail, which is much more complicated. Fortunately, this does not affect the following discussions: since the semantics of specifications are defined by glb and lub, we can always substitute t by a corresponding *limes* expression.

Each *LNode* corresponds to one certain specification particle, i.e. the sub-expression at a certain sequential position of one certain linear specification[5], and to another *LNode* as its *predecessor*.

Each *LNode* n which is in a valid state at a time instant t_{now} represents a (possibly infinite) set of segments, which (1) are the last segment in a partial interpretation which ends at t_{now}, and (2) which fulfill the specification particle corresponding to n.

If n corresponds to the very first particle of a linear specification, each such segments constitutes a partial interpretation on its own.

If not, each such segment constitutes one or more partial interpretations by appending it to one or more partial interpretations represented by the predecessor of n.

If the specification particle corresponding to n is of form ${}^{i,a}p_k$, the node is a `Prime` node. If the specification particle corresponding to n is of form `AND`$\{\}$, the node is an `ASol` node (read: "*sol*ution of an *a*nd expression").

Infinite sets of segments can be represented by a single *LNode* object because of the following reasons:

At each current time instants of it execution, the algorithm relies only on those partial interpretations which end exactly at this very time instant. The same holds for all theoretical discussions related to some arbitrary time instant of the past execution. In this context, the segments of a partial interpretation are uniquely identified by its start time:

The end time of the last segment of each partial interpretation is always identical to the current time instant t_{now}. The end time of a non-last segment in each valid interpretation is identical to the start time of its successor segment.

Therefore only the sets of possible start times have to be implemented to represent a set of segments.

The start times of all segments represented by a valid `Prime` node are determined by applying linear algebraic operations(6.7) on the latest time instant when the corresponding observation function changed to `true`, and (possibly) the time when it changed back to `false`.

Therefore the set of all start times of all segments represented by `Prime` nodes is a cohesive interval, which can be uniquely identified by its lub and glb.

The same holds by induction for segments represented by any *LNode*, because the set of the possible start times of segments represented by an `ASol` node is calculated as an intersection of these intervals.

Since a new `Prime` node is only created when its predecessor node enters its valid state (or possibly once for each negative edge of its observation function, which can happen only finitely often) the collection of currently existing `Prime` nodes is always finite.

[5] This relation from *LNodes* to specification particles is a non-injective function, because different node may refer to the same position in the same linear specification, if they represent different partial interpretations by referring to different predecessors, see the following paragraphs and the last two node objects in figure 4.2.

The same holds by induction for all *LNodes*, because for each combination of valid nodes corresponding to specification particles from a lower syntactical level of nesting, only one `ASol` is created.

4.7.3 Termination of Node Objects

Whenever the algorithm detects in an evaluation step at time instant t_{now}, that none of the partial interpretations represented by a currently valid node n can extend beyond t_{now} this node transits from the valid into the *terminated state*. For the sake of shortness, this will be simply called the *termination* of the node n.

This event can be caused by the expiration of a time-out request, or by an observation function changing to `false`, and is executed in the negative phase of the evaluation step. (5.16)(5.19)(5.46)(5.47)

This event must be signaled to all successor nodes of n (and to all `ASol` nodes using n as part of their solution, see section 4.7.8 below). Receiving this signal will possibly affect the internal states of a successor nodes. (5.51)(5.49)(5.50)(5.53)

In section 5.9 the termination of n is modeled by excluding the corresponding schema from the set $N = GState.nodes$, after this signaling has taken place.[6] This is highlighted by the visual mark-up $\boxed{\setminus\{n\}}$.

Because the functions in the algorithm address nodes only by filters on these sets, e.g. in (5.23) and (5.48), or by the attribute *.predec* if it is known that this predecessor is a valid node and therefore exists in *GState.nodes*, this modeling is consistent.

In the implementation, objects representing terminated nodes are physically deleted by "`mfree()`". This is possible because all parts of their information contents which are required by their successor nodes are cached by dedicated attributes of the latter. This feature is a central achievement of the algorithm and is discussed in detail in section 6.7.

Contrarily, on the level of theoretical discussion in this chapter and in chapter 6, the information contents of "formerly existing" objects is of course accessible.

4.7.4 The Special Node n_{-1} and the Predecessor Relation Seen as a Tree

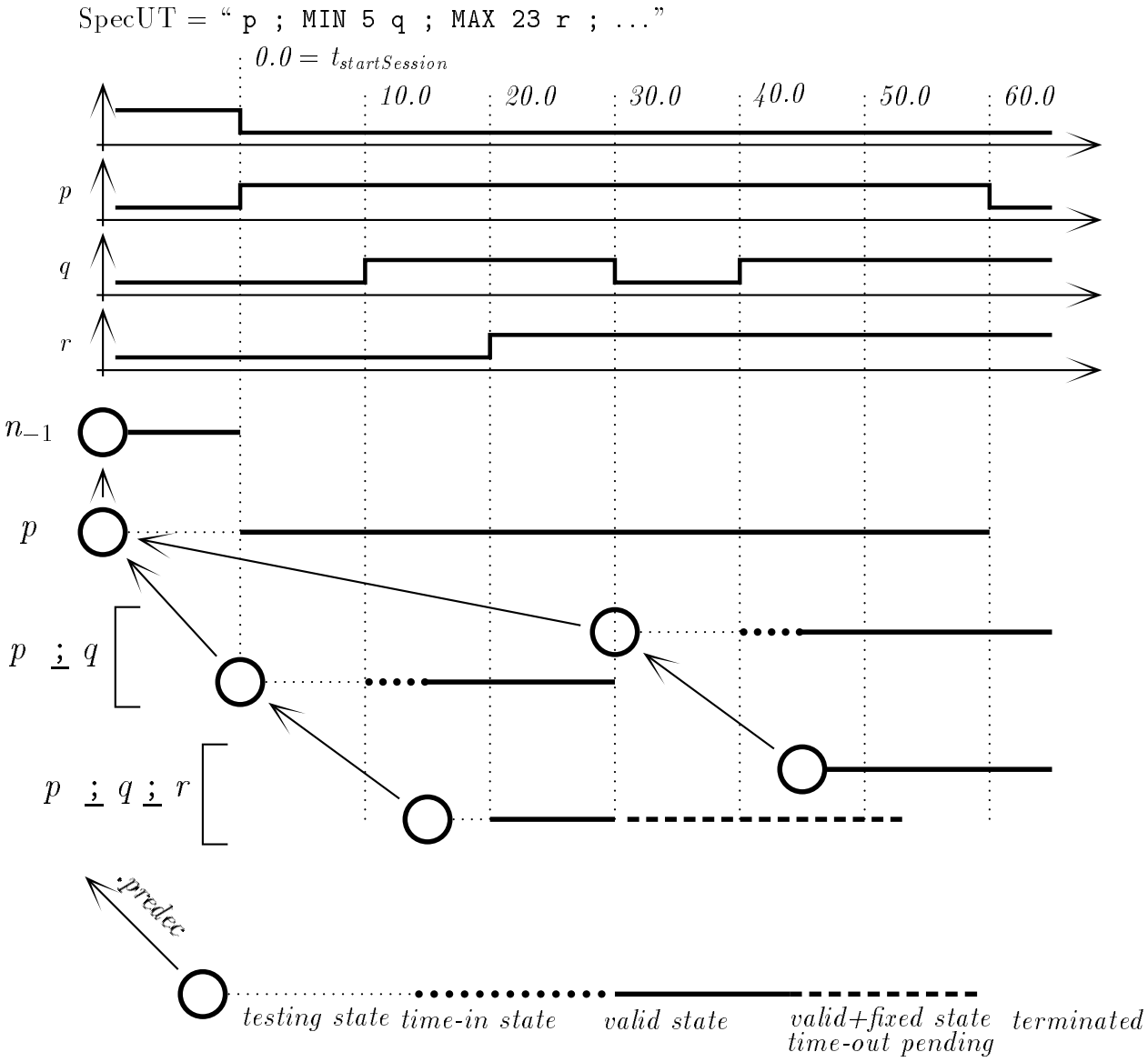
The predecessor function is realized in the algorithm by the node attribute $n.predec$.

For all nodes which represent the very first specification particle in the top level linear specification, the algorithm provides the special `Prime` node n_{-1} to be used as predecessor.

Consequently, the predecessor relation forms a *tree* data structure with n_{-1} as its root. Figure 4.2 shows the top of such a tree, corresponding to the start of a test session.

[6] The name of the schema definition *GState* is used here and in the following as a name of its only instance, which in the algorithm in chapter 5 exists only as the parameter called "g" in numerous function definitions.

Figure 4.2 Example of `Prime` Nodes and Their Predecessor Relation



The node n_{-1} is treated specially by the algorithm (5.8)(5.15): As if it corresponded to an observation function with the meaning "test session has not yet started", it is in a valid state between the calls to *iInit*()(5.8) and the first call to *iNotify*()(5.9).

Since n_{-1} leaves its valid state in the negative phase of the very first evaluation step, each *LNode* which ever reaches a valid state must have as its transitive predecessor an *LNode* which had entered its valid state at the start of the test session.

Therefore each valid node represents a partial interpretation which covers the *total* test session up to now, starting with its beginning. This property is the central goal for the design of the MWATCH algorithm.

4.7.5 Creation of Node Objects for Subsequent Expressions

As explained above, whenever a node n enters a valid state in the positive phase of an evaluation step at time instant t_{now}, this means that there exist partial interpretations of the SUT's trace which extend up to t_{now}.

In the same evaluation step the algorithm must start its monitoring activity, whether partial interpretations (of the SUT's trace up to future time instants) exist, which extend the partial interpretations represented by n by further segments.

The expanded specification expression and the complete set of all linear expressions derivable from SpecUT cannot be realized explicitly in the implementation, due to its possibly infinite cardinality.

Instead, every node n represents by the (inverted) sequence of its predecessor nodes the prefix of a linear specification which is already recognized as being fulfilled by the prefix of the SUT's trace.

Additionally, its attribute $n.expr$ holds the suffix of that sub-expression of the *original* SpecUT, this linear specification has been derived from. So $n.epxr$ is an un-expanded expression $\in \mathcal{L}\, S'$.

$n.epxr$ will be expanded on demand, as soon as n goes valid, calculating for this node the set of *subsequent expressions*. This expansion goes *one step towards* a linear specification by making explicit all choices immediately following the chop operator to the right of the node's own specification particle.

The expanding rules are

$$\begin{array}{lcl} \{(\texttt{OPT}\,\alpha\,\underline{;}\,\beta)\} & \rightsquigarrow & \{(\alpha\,\underline{;}\,\beta),(\beta)\} \\ \{(\texttt{REPst}\,\alpha\,\underline{;}\,\beta)\} & \rightsquigarrow & \{(\alpha\,\underline{;}\,\texttt{REPst}\,\alpha\,\underline{;}\,\beta),(\beta)\} \end{array} \tag{4.2}$$

...and the expansion process is the application of these rules until a fix-point is reached.

Since this expansion process is determined only by the structure of the original specification expression, which is finite, it always terminates.

Let *head of a chop expression* be the left argument of the left-most chop operator contained therein.[7] Then all elements in the set of subsequent expressions are un-expanded expressions from $\mathcal{L}\ S'$, the head of which is always $\in \mathcal{L}\, s\_base$, i.e. either of form $^{\neg\neg}p\_$ or of form $\texttt{AND}\{\ldots\}$.

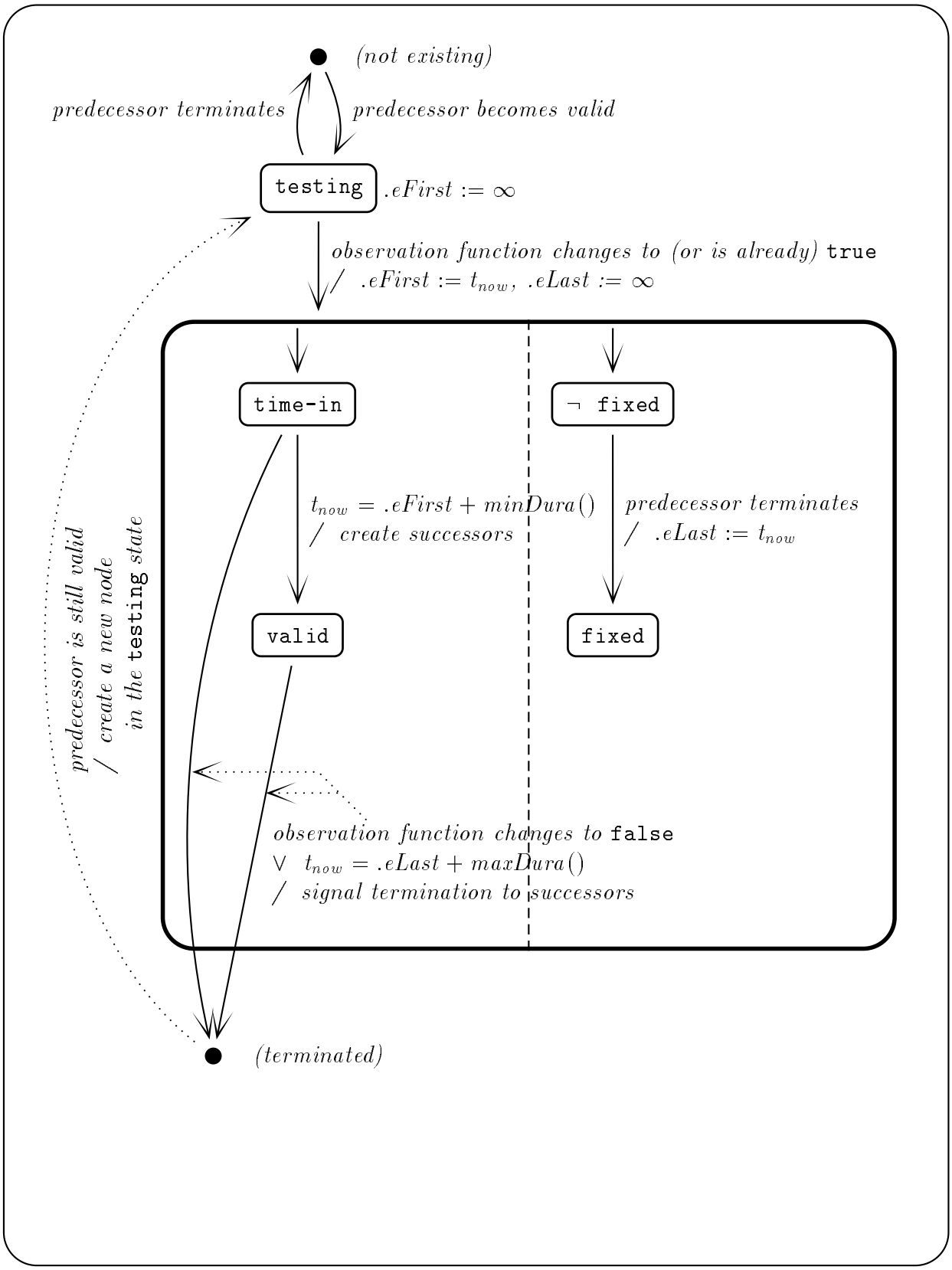
Therefore, in the same evaluation step in which a node becomes valid(5.31) , the set of its subsequent expressions is calculated, and for each element e of this set a new node n_e is created. (5.32)(5.33)(5.35) The value of $n_e.expr$ is set to e, and the value of $n_e.predec$ is set to n, and the head of $n.epxr$ is the specification particle the new node must monitor.

4.7.6 States and Behaviour of `Prime` Nodes

In the case that the head of a subsequent expression e is of form $^{i,a}p_k$, a new `Prime` node n is created(5.33). The further processing of `Prime` nodes is straight-forward,

[7] In the algebraic model in chapter 5 all specification expressions chop expressions, because of the additional appending of the $\triangle_{\triangle}$ symbol, cf. section 4.7.8 below.

Figure 4.3 States of a `Prime` node



and depicted informally by the state machine from figure 4.3.

If the $n.predec$ terminates earlier than that the corresponding observation function v_k has become `true`, $n.predec$ does not represent any partial interpretation which extends beyond than t_{now}. Since segments corresponding to n may not yet begin due to v_k still being `false`, n will never represent any partial interpretation at all. So it is simply deleted. (5.49)

Let t_t either be the time instant of n's creation in case that v_k is already `true` in this moment, or otherwise the subsequent time instant at which v_k changes to true while $n.predec$ is still in a valid state.

At t_t the `Prime` node enters the *time-in state*, and the current time instant is recorded in the node's attribute $.eFirst$ (read "first entry time")(5.20). Since currently its predecessor is still valid, there exist partial interpretations represented by $n.predec$ w.r.t. some linear specification e_P which extend up to t_{now}. In infitely many future time instants $t_t + \varepsilon$ the SUT's trace therefore will surely have fulfilled $e_P \underline{;}^{0.0,\infty} p_k$, i.e. a linear specification which ignores the duration requirements of n's specification particle. This follows from the finite variability requirement imposed on the the input data (cf. section 2.1 on page 4 and discussed in more detail in section 4.7.7 below).

The value of $.eFirst$ indicates the earliest possible time instant at which all those segments corresponding to n's specification particle may begin, which are able to extend one of the partial interpretations represented by $n.predec$.

After the expiration of the minimal duration requirement (5.18)(5.19)(5.21) the node changes to the valid state. It now represents a partial interpretation, because there is at least one segment during which v_k is constantly `true` , which is long enough to fulfill the minimal duration requirement i, and which extends a partial interpretation represented by $n.predec$.

If, otherwise, the observation function changes to `false` again, earlier than the time-in expiration, the node is discarded, because the duration of being-true of the observation function was not long enough to form a valid segment. (5.19)(5.46)(5.47)

Iff, in this case, the predecessor node is still true, the node is re-established in the testing state, waiting for a new segment candidate to begin.(5.46)

When the predecessor of a time-in or valid `Prime` node leaves its valid state in the negative phase of an evaluation step, the time instant of this event is recorded in the field $.eLast$. (5.47)(5.49) This time instant value is the last time instant at which the segment corresponding to n may begin, because all partial interpretations represented by its predecessor node cannot extend beyond this time instant.

Additionally, if there is a maximum duration requirement $a < \infty$ imposed on the specification particle of n, in the same evaluation step a time-out request for the time instant $n.eLast + a$ is initiated(5.49): As soon as this expires, the node terminates (5.19)(5.47), because all segments lasting longer than this current time instant would have a longer duration than permitted, since they must have begun not later than $n.eLast$.

When the observation function of a valid node changes to `false`, then this node transits to the terminated state. (5.19)(5.47).

If its predecessor is still valid, a new node for the same subsequent specification is created, waiting for a new candidate segment to begin. (5.47)

4.7.7 The Kernel Algorithm Needs to Look into the Future !

Whenever a node is created for a specification particle ${}^{i,a}p_k$, and the v_k is currently `true` and does not change to `false` in the same evaluation step, or it changes to `true` in the same evaluation step, the node is put into the time-in state immediately. (5.20) If there is no minimal duration requirement imposed on the specification particle (i.e. $i = 0.0$), then the node transits the time-in state and enters a valid state immediately. (5.21).

In this case, the described process of expanding the set of subsequent expressions and creating the corresponding nodes will be continued recursively in the very same evaluation step, (5.33)(5.34) — until an observation function is referred to wich stays `false` (or which just in this evaluation step changes to `false`), or a specification particle with a non-zero minimal duration requirement is reached.

This behaviour of the implementation is correct because of two facts:

(1) All observation functions which already are (or currently become) `true`can change to `false` only in some future evaluation step. This step cannot happen earlier than after a non-zero time interval, due to the finite variability of the input data (cf. section 2.1 on page 4).

(2) The number of nodes created in the same evaluation step as (transitive) successors of one certain valid node, is always finite: It is limited by the expression SpecUT, which is always a finite term, and all expressions of type `REPst` α (which are the only potential sources of infinite linear specifications) are explicitly prevented from being used for node creation more than once in the same evaluation step and as a consequence of the same node becoming valid. ("live lock prevention", cf. formula (5.33))

Therefore the SUT's trace in that future interval of real-time described in (1) can always be split up into as many segments as nodes have been created.

This even holds if the last of these future nodes corresponds to a specification particle with a duration requirement > 0.0, because this requirement is defined by a *limes* expression, and the interval needed for distributing it to the other future nodes can be made arbitrarily small.

This "looking into the future" is a central contribution to the simplicity of the algorithm, and is also applied to valid `ASol` nodes accordingly.

4.7.8 States and Behaviour of `ATst` and `ASol` Nodes

The central idea to recognize all segments which fulfill a conjunctive expression is to provisionally abstract from the conjunction and to simply replace it by another disjunction:

Whenever a node n_P becomes valid, and an `AND/OR` expression e_a = `AND`$\{$ `OR`$\{\alpha_{1,1}, \ldots, \alpha_{1,k_1}\}, \ldots,$ `OR`$\{\alpha_{n,1}, \ldots, \alpha_{n,k_n}\}$ $\}$ is head of an expression e from

the set of subsequent expressions, all chop sequences $\alpha_{1,1}, \ldots \alpha_{n,k_n}$ are treated as if they were subsequent expressions of n_P on their own right, — as if they were suffices of the top level chop sequence, which specify a complete test session up to its end. (5.38)(5.39)(5.33)

That means that they are partially expanded and a new node is created for each of the expansion results, as described in the previous sections and (recursively) in this section. These nodes are called the *leading nodes* of themselves and of all their successor nodes.[8] For each node n, the path in the predecessor tree which starts with the leading node of n and ends with n is called its containing *node chain*.

Consequently it holds for each node, that its leading node is assigned to the specification particle which is the first in the same chop sequence of the nearest containing `AND/OR` construct as its own specification particle.

For each partial interpretation represented by some node, the *provisional interpretation* is the suffix which starts with the segment represented by the node's leading node.

Not before provisional interpretations have been detected w.r.t. a certain linearization of one *complete* $\alpha_{x,y}$ from *each* `OR` expression, the algorithm re-considers the `AND/OR` expression: All provisional interpretations which start at the same time instant are replaced by one single segment, for extending the partial interpretation of the chop sequence of the next-higher syntactical level.

For this sake, the information which node is a leading node and to which `AND/OR` expression it belongs, is contained in all nodes independently from the node's attributes as discussed so far: For each `AND/OR` construct e_a which is head of an expression e in the set of subsequent specifications of n_P, a new `ATst` node a is created, using n_P as its predecessor. (5.33)(5.35) For each `OR` expression $o_1 \ldots o_n$ contained in e_a, a new `OrGr` object is created, the attribute *tstPartOf* of which refers to a.(5.39)

In all leading nodes created for an $\alpha_{m,\_}$ the attribute *.livesIn* refers to o_m, i.e. the `OrGr` object representing the `OR` expression of which α is an argument. (5.39)(5.33)

In all other cases, i.e. when creating `Prime` nodes as described in the preceding section, or when creating the `ATst` node as described above, the attribute *.livesIn* is simply copied from the predecessor node to all of its successor nodes. (5.32)(5.33) So the leading node of each node is always the first node found when following the predecessor relation which is contained in a different `OrGr` than its predecessor.[9]

Consequently, the value of *.livesIn* of every node always refers to an `OrGr` representing the disjunctive expression which contains in one of its alternatives the node's specification particle. For sake of shortness we will say that each *Node* n is *contained in* n*.livesIn*, and each `OrGr` o is *contained in* o*.tstPartOf*.

[8] ...up to and excluding those successors which are again leading nodes because of a nested `AND/OR` expression contained in an α.

[9] This relation is only used in the discussion and proofs of the algorithm, cf. the final remark in section 4.7.3. In the implementation, the attributes *.seFirst* and *.seLast* of each node n, see below, contain all required information concerning the leading node of n and of all nodes between this and n. Indeed, it is a central achievement of the algorithm that an access to "older" nodes having left their valid state is *not* needed to calculate the correct verdicts, as it is proven in section 6.7.

This includes (naturally) those nodes which correspond to the *last* specification particle of a chop sequence, which is relevant for calculating the set of segments fulfilling the conjunction.

For the sake of uniform treatment, a special `OrGr` object is supplied by the algorithm(5.8), which is in the text referred to by *GState.top*. The attribute *.livesIn* of all nodes which represent a specification particle from the top level chop sequence refers to *GState.top*. *GState.top* is also used for calculating verdicts, see section 4.7.11 below.

The recognition of solutions for `AND/OR` expressions is realized in the algebraic model contained in chapter 5 by a notational transformation: all sub-expressions α of an `AND/OR` expression are extended by chop-wise appending the special terminal symbol $\triangle_\triangle$, i.e. they are transformed into $\alpha \underline{;} \triangle_\triangle$. (5.8)+(5.39) The same transformation applies to the top level expression SpecUT.

Now the algorithm for calculating the set of subsequent expressions of a given node, as defined above, can be re-used: An *LNode* n has reached the end of a linear expansion of an argument of `AND/OR`, iff $\triangle_\triangle$ is contained in its set of subsequent expressions. This fact is reflected by the Boolean attribute *endReached* set to true in the node object. (5.33)(5.40) Such a node is called *final node*.

To extend a partial interpretation represented by the common predecessor n_P (which is a partial interpretation on the next-higher level of the syntactical nesting of `AND/OR` constructs) by a new segment, there must exist an interpretation for this segment's data w.r.t. a linearization of one sub-expression from each `OR` expression, cf. the definition of interpretation in section 4.6.

Each *LNode* object n contains two additional attributes called *.seFirst* and *.seLast* (read "sequence entry first" and "sequence entry last"). These attributes are used to calculate the earliest and the latest time instant, at which all **those** segments corresponding to the leading node of n can begin, which are also members of a partial interpretations represented by n.

These values are **not** just identical with the corresponding values in the partial interpretations represented by the leading node itself. Instead, the correct maintenance of *.seFirst* and, especially, *.seLast* is the crucial point in the design of the algorithm, cf. the following section.

Now the algorithm can work as follows:

In each evaluation step at t_{now} in which the attribute *.endReached* of a node n is changed to true, all combinations of this node with one final node from each other `OrGr` of the same `ATst` node are considered. (5.40) (5.41) (5.43)+(5.42)

For each such combination of nodes, the set of possible start points of those sub-traces is calculated, which extend up to t_{now} and for which each node provides at least one provisional interpretation.(5.42) This calculation is based on the values *.seFirst* and *.seLast* of all combined nodes, and its (possibly infinite) result can be represented by one cohesive interval which consists of all possible start times of these sub-traces, cf. the following section.

As mentioned above, n_P is the common predecessor of all leading nodes and of the `ATst` node.

If the calculated set of sub-traces is non-empty, each sub-trace can be appended to one or more partial interpretations represented by n_P, yielding a new and longer partial interpretation w.r.t. the chop sequence of the next-higher syntactical level.

Since this set of segments is uniquely determined by the cohesive interval consisting of the respective start times (similar as it is with `Prime` nodes), only one single node must be created for its representation.

This node is an `ASol` node. It is put either immediately in a valid state, or it is put into the time-in state, as described in the next section, and enters the valid state when the time-in request expires.

The value of its attribute *.solParts* (read "parts of the solution") identifies the set of final nodes which have led to its creation. The values for *.predec*, *.livesIn* and *.expr* are simply copied from the corresponding `ATst` node.

Therefore, as soon as the `ASol` node becomes valid, the process of calculating the set of subsequent expressions and creating the corresponding nodes is executed in the same way as with a `Prime` node $^{(5.42)(5.31)+(5.19)(5.21)(5.31)}$, as described in the preceding sections, thereby continuing the process of monitoring linear expansions of the specification expression on the next-higher level of syntactical nesting.

An `ASol` node terminates as soon as at least one of the nodes from *.solParts* terminates $^{(5.47)}$, or when a time-out request expires, cf. the next section.

4.7.9 Calculating Duration and Timing Requirements for Segments Representing Solutions of Conjunctions

.

4.7.9.1 Earliest Start Time

As mentioned above, the fundamental semantics of each node currently in a valid state is the fact that there exist partial interpretations of the SUT's trace up to now, w.r.t. some prefix of a linear specification derived from SpecUT.

The node does *not* provide any information about the structure of these partial interpretations: This is neither necessary for the derivation of verdicts, nor technical easily feasible because the complete information is of a cardinality beyond $\mathfrak{c}^{\mathfrak{c}}$.

But *some* information on this structure is needed, namely the set of start times of the provisional suffices, because this set has to be intersected with those of other final nodes for calculating the solutions of conjunctions, as described above.

This set is not identical to the set of possible start times of the segments represented by the leading node. Consider a specification like ...

$$p_1 \ \underline{;} \ \texttt{AND}\{\ \texttt{OR}\{\texttt{MAX}\, d_2\, p_2 \ \underline{;} p_3\},\ \ldots\ \}$$

Assume that v_1 and v_2 stay always `true`.

At the time instant 100.0 also v_3 changes to `true`, and the node n_3 leaves the testing state and enters the valid state.

This indicates that there exist (infinitely many) valid interpretations of the whole trace $D_{[t_{startSession}...100.0+\varepsilon]}$ w.r.t. the specification p_1 ; MAX d_2 p_2 ; p_3.

In spite of v_2 having been valid for a much longer duration, the provisional interpretations which can contribute to a segment fulfilling the conjunction may not begin earlier than $100.0 - d_2$. Since the becoming-valid of v_3 happens not earlier than 100.0, the maximal duration requirement imposed on p_2 would otherwise be violated.

This constraint, the earliest start time of the provisional interpretations represented by n_3 is implemented by the attribute $n_3.seFirst$. This value, in spite of being related to v_2, cannot be realized as an attribute of the node object n_2, because different values may apply to different and simultaneously valid successors of n_2, cf. figure 4.2.

For each node n, the value of $n.seFirst$ only depends from $n.eFirst$ (which is the first possible start time of the node's very own segments) and the sum of all maximal duration requirements imposed on the predecessor nodes in the same node chain. Therefore this value needs only to be calculated once, and stays constant throughout the node's life-time.

For a `Prime` node it is calculated when the node enters the time-in state due to the becoming-true of the corresponding observation function(5.20), which sets the attribute $.eFirst$.

The first possible start time of the segments represented by an `ASol` node is implemented as its attribute $.aeFirst$.[10] When an `ASol` node is created(5.42), this value is set to the latest of the values $.seFirst$ of all the final nodes it combines. In the course of its creation the same calculation for $.seFirst$ is executed as in the case of a `Prime` node. For this purpose the `ATst` node has cached the corresponding value of the common predecessor n_P, since this may have terminated and be deleted in the meantime.

4.7.9.2 Time-In and Time-Out Requests

Consider a specification like ...

$$p_1 \ ; \ \mathtt{AND}\{\ \mathtt{OR}\{\mathtt{MAX}\,7\ p_2\ ; p_3\},\ \mathtt{OR}\{\mathtt{MIN}\,50\ p_4\}\ \}$$

... assuming that v_1, v_2 and v_4 stay always `true`, and at the time instant 100.0 v_3 also changes to `true`. In the evaluation step at 100.0, an `ASol` node N is created, because both sub-expressions are now fulfilled.

Partial interpretations w.r.t. p_1 ; MAX 7 p_2 ; p_3 can start anywhere between 0.0 and 100.0, but the provisional interpretations w.r.t. MAX 7 p_2 ; p_3 cannot start earlier than 93.0, as explained above.

In spite of the minimal duration requirement on p_4 already beeing fulfilled *per se* since 50.0, it has to be re-considered when combining the provisional interpretations

[10] The names $.eFirst$ and $.aeFirst$ has been chosen differently only for documentation purpose, since the calculation of these attributes differs significantly. Seen "from above", in the context of a node chain representing interpretations of a chop sequence, their rôle is identical.

represented by n_3 and n_5: Since the segments represented by N cannot begin earlier than 93.0, the minimal duration requirement on p_4 has to be fulfilled relatively to this time instant, — but only if n_4 is considered in the context of the partial interpretations represented by N.

Therefore each newly created node `ASol` node N is put in the time-in state, iff the longest minimal duration requirement imposed on one of the chop sequences it combines is not yet fulfilled relative to $N.aeFirst$.(5.42)

The same mechanism applies to the shortest maximal duration requirement and $N.aeLast$, possibly generating a time-out request.(5.42)

4.7.9.3 Conflicting Duration Requirements

The transformation from $\mathcal{L}\,S$ to $\mathcal{L}\,S'$ guarantees that $i \leq a$ holds for each specification ${}^{i,a}p_k$ appearing in SpecUT, cf. section 4.2.

But a specification like ...

$$\texttt{AND}\{\ \texttt{OR}\{\texttt{MIN}\,10.0\ p_1,\ \ \texttt{MIN}\,20.0\ p_2\},\ \texttt{OR}\{\texttt{MAX}\,15.0\ p_3,\ \ \texttt{MAX}\,25.0\ p_4\}\ \}$$

...is not analyzed during this transformation in its current implementation.

While three of the four possible combinations of sub-expressions are sensible, the combination of p_2 and p_3 is never satisfiable.

Therefore the function which combines final nodes(5.42) must check for the absence of conflicting duration requirements.

4.7.9.4 Latest Start Time

As described so far, all attributes the values of which are derived from the going `true` of an observation function, stay constant throughout the life-time of a node object: Whenever an observation function changes to `true`, new node objects are created, their attributes are set accordingly and they are just added to the collection of nodes. No attribute values of existing node objects need to be altered.

Naturally this is not true when observation functions return to `false` again: the corresponding node objects already exist, and thus the state of already existing objects has to be altered to reflect this event. This happens e.g. with the attribute $.eLast$ of a valid `Prime` node, when its predecessor node terminates, cf. section 4.7.6. This does not cause any difficulties, because it is a purely local update.

The situation is fundamentally different as soon minimal duration requirements and `AND/OR` expressions are combined:

Consider the following specification:

$$p_1\ \underline{;}\ \texttt{AND}\{\ \texttt{OR}\{p_2\ \underline{;}\ \texttt{MIN}\ d_3\ p_3\ \underline{;}\ p_4\ \underline{;}\ p_5\ \underline{;}\ \texttt{OPT}\,(p_6\ \underline{;}\ p_7)\},\ \beta\ \}\ \underline{;}\ \gamma$$

Assume that all $v_1 \ldots v_7$ stay `true` for a long duration, and the corresponding nodes $n_1 \ldots n_7$ exist in a valid state.

At some time instant t_4, v_4 changes to `false`.

From now on it is clear that the provisional interpretations represented by the final node n_5 may not start later than $t_4 - d_3$, because otherwise the minimal duration constraint imposed on p_3 cannot be fulfilled.

This is reflected in the attribute $.seLast$ of n_5 (read: "sequence entry last"), which for each *LNode* n always holds the time instant for which it is known that provisional interpretations represented by n may not begin later.

Similar as in the dual case described in section 4.7.9.1 above, (1) for each *LNode* the value of $.seLast$ is determined by its value of $.eLast$ and the sum of the minimal duration requirements of all preceding nodes up to the leading node,$^{(5.47)(5.51)+(5.42)}$ and (2) the value of $.aeLast$ of each `ASol` node is defined as the earliest value $.seLast$ of all combined final nodes.$^{(5.42)}$

The difference to the dual case is that already successor nodes n_6 and n_7 have been created. The fact that the value of $n_5.seLast$ has changed must be propagated to all these successor nodes, because it may alter their own value of $.seLast$. This is achieved by the function $LNode\_terminates()$ $^{(5.47)}$ calling $LNode\_SEL\_lowers()^{(5.51)}$ on itself, and recursively on all successor nodes if necessary.

This propagation can only cause an alteration towards a stronger restriction, i.e. a lower value of $.seLast$. In the scenario above, where all observation functions except v_4 are `true` , $n_6.eLast$ and $n_7.eLast$ will be set to $t_4 - d_3$.

But in the case that v_6 has already changed to `false` at some earlier time instant $t_6 < t_4$, then $n_7.seLast$ will already have been altered to $t_6 - d_3$, and the lowering of $n_5.seLast$ to $t_3 - d_3 > t_6 - d_3$ will not have any effect on $n_7.seLast$.

Therefore this propagation will be stopped as soon as it reaches a node on which it has no effect, cf. the definitions of $LNode\_SEL\_lowers()^{(5.51)}$ and $ASol\_subSEL\_lowers()^{(5.52)}$.

Furthermore, one or more solutions of β may have been detected. In this case one or more `ASol` nodes using n_5 as part of the represented solution have already been created, and γ has been expanded and nodes have been created, using these `ASol` nodes as predecessors. Therefore a new, lower value of $n_5.seLast$ must be propagated to all `ASol` nodes N with $n_5 \in N.solParts$, which is achieved by the function $LNode\_SEL\_lowers()^{(5.51)}$ calling $ASol\_subSEL\_lowers()^{(5.52)}$ for all these N.

This possibly alters $N.aeLast$. Any alteration of $N.eaLast$ always changes the value of $N.seLast$, see rule (2) above, so that the propagation process must continue recursively to all successors of N and to all `ASol` nodes which use N as part of their solution. This is achieved by $ASol\_subSEL\_lowers()^{(5.52)}$ calling in turn $LNode\_SEL\_lowers()^{(5.51)}$.

Since a possibly pending time-out request of an `ASol` node N is determined by the constant value $N.maxSubSum$ measured relatively to $N.aeLast$, a lowering of this value must imply a re-adjustment of the timer request.$^{(5.52)}$ The proof that this re-adjustment is always consistent is the main issue in section 6.3.5.1.

This propagation mechanism constitutes the major part of the activities in the negative phase of an evaluation step. Its definition and the proof of its correctness are the central invention in the design of the *MW*ATCH algorithm.

4.7.10 Optimization of the Monitoring of Conjunctive Expressions

The process of monitoring the fulfillment of a conjunctive expression, as described in the preceding section, is slightly modified for the sake of performance as follows:

Each sub-trace of the SUT's trace which fulfills a conjunctive expression cannot begin earlier than a time instant at which the corresponding combination of those observation functions takes the value true, which are the leading atomic predicates in all combined chop sequences.

E.g. the nested specification (given in the front-end notation S) ...

$$\begin{array}{l} \texttt{AND}\{\ (p_1\ \underline{;}\ldots),\\ \qquad \texttt{OR}\{\ (p_2\ \underline{;}\ldots),(p_3\ \underline{;}\ldots),\ \texttt{AND}\{\ (p_4\ \underline{;}\ldots),(p_5\ \underline{;}\ldots)\}\ \}\\ \quad\ \}\end{array}$$

...can never map a sub-trace of the SUT's trace which does not at its begin fulfill the observation function ...

$$p_1\ \wedge\ (\ p_2\ \vee\ p_3\ \vee\ (p_4 \wedge p_5)\ )$$

Therefore the rewriting from S to S', executed in the pre-processing step of the adaptive layer and described in section 4.2 above, synthesizes an additional observation function and a corresponding atomic predicate for each conjunctive expression. With this atomic predicate p_k the `AND` expression is attributed, which is written in the syntax of S' as an index $\texttt{AND}_k$, cf. table 4.1 above.

In the concrete implementation, the process of node creation corresponding to a conjunctive expression, as described in the preceding paragraph, is suspended after the creation of the `ATst` node [5.33]. The further operations (creation of the `OrGr` objects and of the nodes for the sub-expressions contained in the disjunctions contained in the conjunction) are not executed until this observation function v_k changes to `true`. [5.19][5.20][5.38]

This feature contributes significantly to the efficiency, because the monitoring of all sub-expressions of a conjunction can be a rather costly process. If it were started earlier, it would only detect solutions of these sub-expressions which would (at least partly) be discarded anyway when constructing the conjunctive combinations, or, even worse, never lead to a valid combination.

4.7.11 Deriving Verdicts

An `OrGr` object which does not contain a single *LNode* in the testing, time-in or in a valid state, nor a single `ATst` node, it is called *futile*.

A futile `OrGr` will nevermore produce nodes which represent partial interpretations.

Therefore, as soon as one of its `OrGr` objects goes futile because the last *LNode* node object it containes has terminated, (5.47)(5.53) the `ATst` node to which this `OrGr` belongs, all of its `OrGr` objects and all recursively contained nodes therein are deleted. (5.53)(5.54)

This may have the effect that the `OrGr` having contained the `ATst` node does not contain a single node anymore and becomes futile itself, so that the process of deletion continues recursively.(5.53).

If the top level `OrGr` object *GState.top* becomes futile, the SpecUT cannot be fulfilled any more. In this case, an early `fail` verdict is returned by the algorithm.(5.11)

Contrarily, as soon as *GState.top* contains a valid node n representing the specification particle ${}^{i,a}p_0$ (i.e. a node representing the `ANY` construct from $\mathcal{L}\,S$, which corresponds to the observation function v_0 which is always `true`) and if the maximal duration constraint a is either equal to ∞, or it is smaller, but the time instant of the time-out request is beyond the known latest end of the test session, then an early `pass` verdict is returned by the algorithm, because the SUT's behaviour will always fulfill that linear specification represented by n.(5.11)

If the end of the test session is reached, and the top level `OrGr` referred to by *GState.top* contains any final node (i.e. a node which has reached the end of one linearization of SpecUT), then the final verdict is `pass`, otherwise it is `fail`.(5.12)

Chapter 5

Definition of the Kernel Algorithm

5.1 Structure of this Chapter

In this chapter the kernel algorithm is presented in several sections.

- Section 5.3 defines the types of data the algorithm works on.
- Section 5.4 defines the interface functions which are called by the adaptive layer.
- Section 5.5 defines the top level scheduling functions, which call the functions of the positive and the negative phases on the appropriate node objects, and which schedule those evaluation steps which have not been triggered by a call to *iNotify*(), because they are related only to internal timer expirations.
- Section 5.7 defines the reactions on the becoming-true of observation functions and the expiring of min-timers, which cause nodes to enter a valid state, and which lead to the creation of new node objects.
- Section 5.8 defines the special activities for calculating the solutions of conjunctions. This section and the preceding one define the positive phase of each evaluation step.
- Section 5.9 defines the negative phase of an evaluation step, containing the reactions on the becoming-false of observation functions and the expiration of max-timers.

5.2 Principles of Modeling and Notation

The formulæ contained in this chapter constitute the *MW*ATCH kernel algorithm.

In contrast to its object-oriented implementation in the existing *MW*ATCH tool, which uses C++ and high-order data structures like sets and maps from the *Standard Template Library* (STL), the modeling contained herein is purely functional.

The functions realizing the positive and the negative phase of an evaluation step operate on a set of objects, containing *Node* objects and the auxiliary `OrGr` objects. This set is called N in the rest of this section.

Objects are modeled as schema values, i.e. elements of a product type with named components, similar to the notion of schema known from the Z language.[19]

The names of the schema definitions are used as constructor functions, taking a list of assignments which give the initial values of the named components, like `OrGr`($exprX = expr$) in formula (5.8).

The modeling of the object collection as a mathematical set of expressions is feasible because the combination of predecessor node, corresponding specification particle and $.eFirst$ is unique for all simultaneously existing node objects, and therefore the corresponding algebraic expressions are always distinguishable, and they uniquely identify the data object from the imperative implementation.[1]

In contrast to the implementation, where the inter-node relations are realized by bi-directional references, partly involving `set` and `map` objects from the STL, in the algebraic model the references are strictly ordered: a newly created object only refers to "older" objects. The graph of all inter-node relations is thus non-cyclic. The inverse relations are modeled by explicitely applying inquiry functions to the complete set of nodes N (like $usedInSolution()$ and $successors()$ in formula (5.48) etc.).

Therefore references to other node objects (as established by the values of `predec` and `solParts`) can be modeled by simply repeating the corresponding schema values, preserving the finiteness of N seen as an algebraic term.

The transformations on N are realized as follows:

The notation

$$s' = (s \oplus f_1 = e_1;\ f_2 = e_2;\ \ldots)$$

means the derivation of a new schema value s' by overriding the values of the fields $f_1, f_2, \ldots$ by the given expressions[2], while copying the values of all other fields which are not explicitely assigned a new value. This derivation is purely functional.

The first modifications in the life-time of a node object (entering the testing, a time-in and a valid state) include the corresponding alterations of the node's data state. These alterations are purely *local* updates, since, as long as node is not yet in a valid state, no other node can exist which has a reference to this object.

[1] The field $.exprX$ has been added to the definition of the `OrGr` objects only to allow this kind of modeling.

[2] If an attribute $f_1, f_2, \ldots$ appears in such an expression $e\_$ on the right side of an initilization, it refers to the value currently valid in s, i.e. the "old" value.

Therefore the alteration of the global object set is just the exchange of its current member n_o with the new, modified schema instance n_n. This is written as

$$N \diamond n_n \div n_o \stackrel{def}{=} (N \setminus \{n_o\}) \cup \{n_n\}$$

As soon as successor nodes are installed, or the node is used in the construction of an `ASol` node, there do exist references to the node object.

The data state of a valid node object n will alter only if the observation function corresponding to itself or to one of its predecessor nodes changes to `false`. In the implementation, the necessary local alterations are realized by simply updating the values of some fields of the corresponding "`struct`" object of the C++ language.

In the algebraic representation used herein, the modification has to be performed in all terms contained in the node set which contain sub-terms representing the reference to n.

Again, since the structure of the relations is strictly ordered and non-cyclic, these global modifications can be precisely modeled in a pure functional way, by application of the function

$$N \bigstar n_n \div n_o$$

This function is defined in formula (5.45) and performs a recursive visiting of all node expressions which represent successor nodes and `ASol` nodes referring to the node n by some attribute value. In contrast to its recursive definition, its application can simply be read as "exchange n_o by n_n globally in the set of node objects".

While this function performs the same visiting sequence as the algorithms in section 5.9, both have by intention *not* been unified: The former is just a "modeling trick" for representing the object oriented local update in the world of algebraic terms and is not found in the implementation, while the latter are visitor operations which are really executed in both worlds.

As mentioned above, the algorithm is presented in several sections.

The definitions of all central operational functions (i.e. the functions which perform transformations on the object set) which are exported by the containing text section and applied in formulæ contained in other sections are marked by framing their identifier. The application of an operational function which is defined in another section is marked by underlining.

A similar mark-up is used for the definition and the application of auxiliary functions which are used in more than one section. These functions only derive values from the current state of the object set and do not execute any modifications.

5.3 Data Types

$$
\begin{aligned}
Verdicts &= \{\,\texttt{pass}, \texttt{fail}, \texttt{inconc}\,\} \qquad (5.1)\\
\mathbb{D}_+ &= \mathbb{D} \cup \{\infty\}\\
\mathbb{T}_+ &= \mathbb{T} \cup \{\infty\}
\end{aligned}
$$

$$
\begin{aligned}
Values_+ &= \{p_1, \ldots, p_{GState.pCount}\} \rightarrow Boolean \qquad (5.2)\\
Values &= \{p_0, p_1, \ldots, p_{GState.pCount}\} \rightarrow Boolean\\
internalize(v : Values_+) : Values &= v \cup (p_0 \mapsto \texttt{true})
\end{aligned}
$$

$$
\begin{aligned}
GState = \textbf{struct}\ \{\ & nodes : && ObjSet\\
& top : && \texttt{OrGr}\\
& n_{-1} : && LNode\\
& pCount : && \mathbb{N}\\
& firstcall : && Boolean\\
& v_{old} : && Values\\
& t_{startSession} : && \mathbb{T}\\
& sessionLimit : && \mathbb{D}_+\\
\}\ & && \qquad (5.3)
\end{aligned}
$$

$$
\begin{aligned}
EState = \textbf{struct}\ \{\ & nodes : && ObjSet\\
& values : && Values\\
& now : && \mathbb{T}\\
& visited : && \mathcal{L}\ S'\\
\}\ & && \qquad (5.4)
\end{aligned}
$$

The special value $\infty : \mathbb{T}_+$ is used as the time instant value of events which have not yet happened (e.g. $n.eLast = \infty$ indicates that the predecessor node has not yet gone invalid) or which are not scheduled to happen (e.g. $n.toPending = \infty$ means that no time-out timer request is currently active for the node n).

For this value the following operations are defined:

$$
\begin{aligned}
&\forall\, t \in \mathbb{T} \bullet \qquad (5.6)\\
&\quad \infty + t = \infty\\
&\quad \infty - t = \infty\\
&\quad t < \infty = \texttt{true}
\end{aligned}
$$

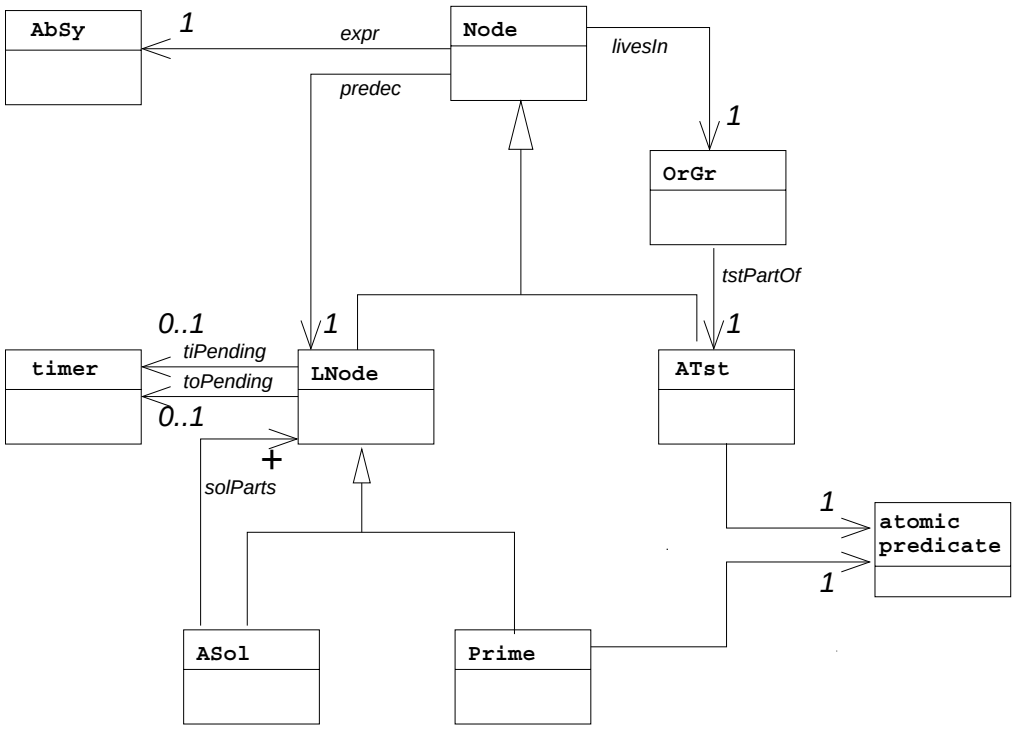
Further combinations are used in the formulæ neither of this nor of the following chapter.

The special value $\infty : \mathbb{D}_+$ represents an unspecified maximal duration requirement, and it is used as a value for the parameter *maxSessionDuration* to *iInit*() if the maximal session duration is not known in advance.

To $\infty : \mathbb{D}_+$ and $\mathbb{D}$ the same arithmetic rules apply as to $\infty : \mathbb{T}_+$ and $\mathbb{T}$, respectively.

The functions earliest() and latest() take a collection of time instants and deliver the minimum resp. maximum value contained in this set. The wording has been chosen for the sake of intuition, esp. in the proof discussions contained in chapter 6.

Figure 5.1 The Object World of the Evaluating Machine in Graphical Notation



An instance of the schema *GState* models the global state the algorithm works on. Its main component *.node* is the above-mentioned collection of *Node* objects and auxiliary `OrGr` objects. Further on *GState* contains constant configuration parameters, and a cache for the values the observation functions have taken at the last call of *iNotify*(). This set is needed for those evaluation steps caused by timer expirations which have not yet been executed between the last and the current call of *iNotify*, as described in section 4.4 above.

The data type *EState* is an auxiliary data type which all transformations in the positive phase of the evaluation step operate on. It simply bundles the parameters *.now* and *.values*, the "in-out-parameter" *.nodes* and an accumulator of already installed `REPst`-expressions (=*.visited*) which is needed for preventing life-locks.

$$
\begin{array}{lcrll}
\texttt{Node} & = & \textbf{struct}\ \{ & \mathit{expr}: & \textbf{opt}^3\ \mathcal{L}\,S' \\
 & & & \mathit{predec}: & \textbf{opt}^3\ \mathit{LNode} \\
 & & & \mathit{livesIn}: & \textbf{opt}^3\ \mathit{OrGr} \\
 & & \} & & \\
\texttt{LNode} & = & \texttt{Node} + \textbf{struct}\ \{ & \mathit{seFirst}, \mathit{seLast}: & \mathbb{T}_+ \\
 & & & \mathit{sumMinPreds}: & \mathbb{D} \\
 & & & \mathit{sumMaxPreds}: & \mathbb{D}_+ \\
 & & & \mathit{tiPending}, \mathit{toPending}: & \mathbb{T}_+ \\
 & & & \mathit{endReached}: & \mathit{Boolean} \\
 & & \} & & \\
\texttt{Prime} & = & \texttt{LNode} + \textbf{struct}\ \{ & \mathit{eFirst}, \mathit{eLast}: & \mathbb{T}_+ \\
 & & & \mathit{pNum}: & \mathbb{N} \\
 & & \} & & \\
\texttt{ASol} & = & \texttt{LNode} + \textbf{struct}\ \{ & \mathit{solParts}: & \textbf{set of}\ \mathit{LNode} \\
 & & & \mathit{aeFirst}: & \mathbb{T} \\
 & & & \mathit{aeLast}: & \mathbb{T}_+ \\
 & & & \mathit{minSubSum}, & \\
 & & & \mathit{maxSubSum}: & \mathbb{D}_+ \\
 & & \} & & \\
\texttt{ATst} & = & \texttt{Node} + \textbf{struct}\ \{ & \mathit{pNum}: & \mathbb{N} \\
 & & & \mathit{seFirstCache}, \mathit{seLastCache}: & \mathbb{T}_+ \\
 & & \} & & \\
\texttt{OrGr} & = & \textbf{struct}\{ & \mathit{tstPartOf}: & \textbf{opt}^4\ \texttt{ATst} \\
 & & & \mathit{exprX}: & \mathcal{L}\,S' \\
 & & \} & &
\end{array}
\tag{5.6}
$$

$$
\begin{array}{lcl}
\mathit{LNode} & = & \texttt{Prime} \cup \texttt{ASol} \\
\mathit{RNode} & = & \texttt{Prime} \cup \texttt{ATst} \\
\mathit{Node} & = & \texttt{Prime} \cup \texttt{ASol} \cup \texttt{ATst} \\
\mathit{Object} & = & \mathit{Node} \cup \texttt{OrGr} \\
\mathit{ObjSet} & = & \textbf{set of}\ \mathit{Object}
\end{array}
\tag{5.7}
$$

[3]Only for the pseudo-node n_{-1}, representing the meta-condition "test session has not yet started", this fields may and must be = `null`.

[4]Only for the special `OrGr`-object *GState.top* this field may and must be = `null`.

5.4 Interface Functions

$$
\begin{array}{l}
\boxed{iInit}\,(expr : \mathcal{L}\,S', predicateCount : \mathbb{N}, maxSessionDuration : \mathbb{D}_+) : GState \\
\quad = \; GState\;(\; top \;=\; \texttt{OrGr}(\,exprX = expr\,) \\
\qquad\qquad n_{-1} \;=\; LNode(\,expr = predec = livesIn = \texttt{null}\;) \\
\qquad\qquad pCount \;=\; predicateCount \\
\qquad\qquad firstCall \;=\; true \\
\qquad\qquad sessionLimit \;=\; maxSessionDuration \\
\qquad\qquad nodes \;=\; E_2.nodes \\
\qquad\quad) \\
\quad \textbf{where} \;\; allFalse : Values \;=\; \{(p_0 \mapsto \texttt{false}), \ldots, (p_{predicateCount} \mapsto \texttt{false})\} \\
\qquad\qquad E_1 \;=\; EState\,(\,nodes = \{n_{-1}, top\}, values = allFalse, \\
\qquad\qquad\qquad\qquad visited = \{\}, now = \bot) \\
\qquad\qquad E_2 \;=\; \underline{termInst}\,(E_1, n_{-1}, top, (expr\;\underline{;}\;\triangle_\triangle)\,)
\end{array}
\tag{5.8}
$$

$$
\begin{array}{l}
\boxed{iNotify}\,(G : GState, now : \mathbb{T}, v : Values_+) : GState \times Verdict \times \mathbb{T}_+ \\
\quad = \; (G_3, deriveVerdict(G_2), t_{next}) \\
\quad \textbf{where} \;\; v_1 \;=\; internalize(v) \\
\qquad\qquad G_1 \;=\; G \oplus (t_{startSession} = now;\; firstCall = \texttt{false}) \\
\qquad\qquad G_2 \;=\; \textbf{if}\; G.firstCall\; \textbf{then}\;\; \underline{evalNodes\_initial}\,(G_1, t, v_1) \\
\qquad\qquad\qquad\quad \textbf{otherwise}\qquad\quad \underline{evalNodes}\,(G, t, v_1) \\
\qquad\qquad G_3 \;=\; G_2 \oplus (v_{old} = v_2) \\
\qquad\qquad t_{next} \;=\; \mathsf{earliest}\,(\;\textbf{map}\;(G_2.nodes, \_.tiPending) \\
\qquad\qquad\qquad\qquad \cup\, \textbf{map}\;(G_2.nodes, \_.toPending) \;\cup\; \{\infty\}\,)
\end{array}
\tag{5.9}
$$

$$
\begin{array}{l}
\boxed{iFinalize}\,(G : GState, now : \mathbb{T}) : Verdict \setminus \{\texttt{inconc}\} \\
\quad = \; deriveVerdict\_final(\underline{evalNodes\_final}\,(G))
\end{array}
\tag{5.10}
$$

$$
\begin{array}{l}
deriveVerdict(G : GState) : Verdict \\
\quad = \; \textbf{if}\; G.top = \{\}\; \textbf{then} \qquad\qquad\qquad\qquad \texttt{fail} \\
\qquad\quad \textbf{if}\;\; \exists\, n \in finalNodes(G.top) \\
\qquad\qquad \bullet\; n.expr = \neg^a p_0 \\
\qquad\qquad \wedge\;\; te = G.t_{startSession} + G.sessionLimit < \infty \\
\qquad\qquad \wedge\; \mathsf{earliest}\,(t_{now}, n.eLast) + a \geq te\; \textbf{then} \qquad \texttt{pass} \\
\qquad\quad \textbf{otherwise} \qquad\qquad\qquad\qquad\qquad\qquad \texttt{inconcl}
\end{array}
\tag{5.11}
$$

$$
\begin{array}{l}
deriveVerdict\_final(G : GState) : Verdict \setminus \{\texttt{inconc}\} \\
\quad = \; \textbf{if}\; finalNodes(G.top) = \{\}\; \textbf{then}\;\; \texttt{fail} \\
\qquad\quad \textbf{otherwise} \qquad\qquad\qquad\qquad \texttt{pass}
\end{array}
\tag{5.12}
$$

$$
\begin{array}{l}
\boxed{finalNodes}\,(N : ObjSet, o : \texttt{OrGr}) : \textbf{set of}\; LNode \\
\quad = \; \{n \in allInhabitants(N, o) \cap LNode \mid n.endReached = \texttt{true}\}
\end{array}
\tag{5.13}
$$

$$
\begin{array}{l}
\boxed{allInhabitants}\,(N : ObjSet, o : \texttt{OrGr}) : \textbf{set of}\; Node \\
\quad = \; \{n \in N \cap Node \mid n.livesIn = o\}
\end{array}
\tag{5.14}
$$

5.5 Top Level Scheduling Functions

$$
\begin{array}{l}
\boxed{evalNodes\_initial}(g : GState, t : \mathbb{T}, v : Values) : GState \;=\; g \oplus (nodes = N_2) \\
\quad \textbf{where} \;\; N_1 = execute\_goTrue(g.N, t, v, goingActive(g.N, v)) \\
\qquad\qquad\;\; N_2 = execute\_goFalse(N_1, t, \{g.n_{-1}\})
\end{array}
\tag{5.15}
$$

$$
\begin{array}{l}
\boxed{evalNodes}(g : GState, t : \mathbb{T}, v : Values) : GState \;=\; g \oplus (nodes = N_4) \\
\quad \textbf{where} \;\; N_1 = execute\_minMax(g.N, t, g.v_{old}) \\
\qquad\qquad\;\; N_2 = execute\_goTrue(N_1, t, v, goingActive(N_1, v)) \\
\qquad\qquad\;\; N_3 = execute\_goFalse(N_2, t, goingInactive(N_2, v)) \\
\qquad\qquad\;\; N_4 = execute\_max(N_3, t, timingOut(N_3, t))
\end{array}
\tag{5.16}
$$

$$
\begin{array}{l}
\boxed{evalNodes\_final}(g : GState, t : \mathbb{T}) : GState \;=\; g \oplus (nodes = N_1) \\
\quad \textbf{where} \;\; N_1 = execute\_minMax(g.N, t, g.v_{old})
\end{array}
\tag{5.17}
$$

$$
\begin{array}{l}
execute\_minMax(N : ObjSet, t : \mathbb{T}, v : Values) : ObjSet \\
\quad = \;\; \textbf{if } ti \leq to \;\wedge\; ti \leq t \textbf{ then} \quad execute\_minMax(N_1, t, v) \\
\qquad\;\; \textbf{if } ti > to \;\wedge\; to < t \textbf{ then} \quad execute\_minMax(N_2, t, v) \\
\qquad\;\; \textbf{otherwise} \qquad\qquad\qquad\quad N \\
\quad \textbf{where} \;\; ti \;=\; \mathsf{earliest}\,(\mathbf{map}(N, \_.tiPending) \cup \{\infty\}) \\
\qquad\qquad\;\; to \;=\; \mathsf{earliest}\,(\mathbf{map}(N, \_.toPending) \cup \{\infty\}) \\
\qquad\qquad\;\; N_1 \;=\; execute\_min(N, ti, v, timingIn(N, ti)) \\
\qquad\qquad\;\; N_2 \;=\; execute\_max(N, to, v, timingOut(N, to))
\end{array}
\tag{5.18}
$$

$$
\begin{array}{l}
execute\_min(N : ObjSet, t : \mathbb{T}, v : Values, NI : ObjSet) : ObjSet \\
\quad = \; \textbf{reduce}\,(NI, LNode\_timeIn(\_, t, v, \_), N) \\
execute\_max(N : ObjSet, t : \mathbb{T}, NO : ObjSet) : ObjSet \\
\quad = \; \textbf{reduce}\,(NO, \underline{LNode\_terminates}(\_, t, \_), N) \\
execute\_goTrue(N : ObjSet, t : \mathbb{T}, v : Values, NT : ObjSet) : ObjSet \\
\quad = \; \textbf{reduce}\,(NT, RNode\_signalRaises(\_, t, v, \_), N) \\
execute\_goFalse(N : ObjSet, t : \mathbb{T}, NF : ObjSet) : ObjSet \\
\quad = \; \textbf{reduce}\,(NF, \underline{Prime\_terminates}(\_, t, \_), N)
\end{array}
\tag{5.19}
$$

$$
\begin{array}{l}
\boxed{RNode\_signalRaises}(N : ObjSet, t : \mathbb{T}, v : Values, n : LNode) : ObjSet \\
\quad = \ \textbf{if}\ n \in \texttt{ATst}\ \textbf{then} \qquad \underline{ATst\_installParts}(E, n).nodes \\
\qquad \textbf{if}\ n \in \texttt{Prime} \ \wedge \ minDura(n) > 0.0\ \textbf{then} \\
\qquad\quad N \diamond (n' \oplus tiPending = t + minDura(n)) \div n \\
\qquad \textbf{if}\ n \in \texttt{Prime} \ \wedge \ minDura(n) = 0.0\ \textbf{then} \quad LNode\_timeIn(N, t, v, n') \\
\quad \textbf{where}\ \ E \ = \ EState(now = t, values = v, visited = \{\}, nodes = N) \\
\qquad\qquad n' \ = \ n \oplus eFirst = t;\ seFirst = \\
\qquad\qquad\qquad\qquad \textbf{if}\ isLeading(n) \ \ \textbf{then}\ t \\
\qquad\qquad\qquad\qquad \textbf{otherwise} \qquad \mathsf{latest}(n.predec.seFirst, t - n.sumMaxPreds)
\end{array}
\tag{5.20}
$$

$$
\begin{array}{l}
LNode\_timeIn(N : ObjSet, t : \mathbb{T}, v : Values, n : LNode) : ObjSet \ = \ N_2 \\
\quad \textbf{where}\ \ N_1 \ = \ N_1 \diamond (n \oplus (tiPending = \infty)) \div n \\
\qquad\qquad E \ = \ EState(now = t, values = v, visited = \{\}, nodes = N_1) \\
\qquad\qquad N_2 \ = \ \underline{LNode\_becomesValid}(E, n).nodes
\end{array}
\tag{5.21}
$$

$$
\begin{array}{l}
\boxed{minDura}(n : \texttt{Prime}) : \ \mathbb{D} \ = \ i\ \textbf{where}\ n.expr = {}^{i,-}p\_\underline{;}\ e \\
\boxed{minDura}(n : \texttt{ASol}) : \ \mathbb{D} \ = \ n.minSubSum
\end{array}
\tag{5.22}
$$

$$
\begin{array}{l}
goingActive(N : ObjSet, v : Values) : \ \textbf{set of}\ RNode \\
\quad = \ \{n \in N \cap RNode \mid v[n.pNum] = \texttt{true} \wedge isTesting(N, n)\} \\
goingInactive(N : ObjSet, v : Values) : \ \textbf{set of}\ \texttt{Prime} \\
\quad = \ \{n \in N \cap \texttt{Prime} \mid v[n.pNum] = \texttt{false} \wedge isActive(n)\} \\
timingIn(N : ObjSet, t : \mathbb{T}) : \ \textbf{set of}\ LNode \ = \ \{n \in N \cap LNode \mid n.tiPending = t\} \\
timingOut(N : ObjSet, t : \mathbb{T}) : \ \textbf{set of}\ LNode \ = \ \{n \in N \cap LNode \mid n.toPending = t\}
\end{array}
\tag{5.23}
$$

$$
isActive(n : \texttt{Prime}) : Boolean \ = \ (n.eFirst \neq \infty)
\tag{5.24}
$$

$$
\begin{array}{lll}
\boxed{isTesting} & : & \textbf{set of}\ LNode \times LNode \rightarrow Boolean \\
isTesting(N, n : \texttt{Prime}) & = & \neg \ isActive(n) \\
isTesting(N, n : \texttt{ATst}) & = & (\ tstParts(N, n) = \{\}\ )
\end{array}
\tag{5.25}
$$

$$
\begin{array}{lll}
\boxed{tstParts}(N, n) & : & \textbf{set of}\ \texttt{OrGr} \\
 & = & \{o \in N \cap \texttt{OrGr} \mid o.tstPartOf = n\}
\end{array}
\tag{5.26}
$$

$$
\boxed{isLeading}(n : Node) : Boolean \ = \ n.livesIn \neq n.predec.livesIn
\tag{5.27}
$$

5.6 Internal Auxiliary Functions

$$
\begin{array}{l}
\textbf{reduce}\,(s : \textbf{set of}\, S,\, f : S \times V \to V,\, v : V) : V \\
\quad = \quad \textbf{if}\, s = \{\}\ \textbf{then}\quad v \\
\qquad\quad \textbf{otherwise}\qquad \textbf{reduce}\,(s \setminus \{t\},\, f,\, f(t, v)) \\
\qquad\quad \textbf{where}\ t \in s
\end{array}
\tag{5.28}
$$

$$
\begin{array}{l}
\textbf{map}\,(s : \textbf{set of}\, S,\, f : S \to T) : \textbf{set of}\ T \\
\quad = \ \textbf{reduce}\,(s,\, \_ \cup \{f(\_)\},\, \{\})
\end{array}
\tag{5.29}
$$

$$
\begin{array}{rcl}
N \diamond n_2 \div n_1 & : & \mathit{ObjSet} \times \texttt{Prime} \times \texttt{Prime} \to \mathit{ObjSet} \\
& = & (N \setminus \{n_1\}) \cup \{n_2\}
\end{array}
\tag{5.30}
$$

In the following text, the function-valued argument passed to **reduce** and **map** is written simply as an expression containing place holders, e.g. "$\_ \cup \{\_\}$". The assignment of arguments to parameters is always uniquely defined by their types.

5.7 Nodes Entering a Valid State

$$
\boxed{\mathit{LNode\_becomesValid}}\,(s : \mathit{EState}, n : \mathit{LNode}) : \mathit{EState} \ = \ \mathit{nextInst}(e, n, n.\mathit{expr})
\tag{5.31}
$$

$$
\begin{array}{l}
\mathit{nextInst}(s : \mathit{EState}, n : \mathit{LNode}, e : \mathcal{L}\,S') : \mathit{EState} \\
\quad = \quad \textbf{if}\ \mathit{hd} \in \mathcal{L}\,\texttt{REPst} \ \wedge\ e \notin s.\mathit{visited} \\
\qquad\qquad \textbf{then}\quad \mathit{terminst}(s \oplus (\mathit{visited} = s.\mathit{visited} \cup \{e\}),\, n, n.\mathit{livesIn}, e) \\
\qquad\quad \textbf{otherwise}\quad \mathit{terminst}(s, n, n.\mathit{livesIn}, \mathit{tl}) \\
\ \ \textbf{where}\ e \ = \ \mathit{hd}\ \underline{;}\ \mathit{tl} \ \wedge\ \mathit{hd} \notin \mathcal{L}\,\underline{;}
\end{array}
\tag{5.32}
$$

$$
\begin{array}{ll}
\boxed{\mathit{termInst}}\,(s : \mathit{EState}, n : \mathit{LNode}, o : \texttt{OrGr}, e : \mathcal{L}\,S') : \mathit{EState} & \\
\quad = \quad \textbf{if}\ e \ = \ \triangle_E\,\textbf{then} & \mathit{termInst}(s, n, o, E) \\
\qquad\quad \textbf{if}\ e \ = \ \triangle_\triangle\,\textbf{then} & \underline{\mathit{LNode\_endReached}}(s, n) \\
\qquad\quad \textbf{if}\ e \ = \ \texttt{OPT}\,E\ \underline{;}\ F\ \textbf{then} & \mathit{termInst}(\mathit{termInst}(s, n, o, F), n, o, E\,\underline{;}\,F) \\
\qquad\quad \textbf{if}\ e \ = \ (\texttt{REPst}\,E)\ \underline{;}\ F\ \textbf{then} & \mathit{termInst}(\mathit{termInst}(s, n, o, F), n, o, E\,\underline{;}\,\triangle_e) \\
\qquad\quad \textbf{otherwise} & \mathit{//\ i.e.}\ e \ = \ \texttt{AND}\{\}\,\underline{;}\,F \ \vee\ e \ = \ \neg\neg p\,\underline{;}\,F \\
& \mathit{checkNewActive}(s \oplus (\mathit{nodes} = \mathit{nodes} \cup \{N\}), N) \\
& \textbf{where}\ N \ = \ \mathit{newNode}(n, o, e)
\end{array}
\tag{5.33}
$$

$$
\begin{array}{ll}
\mathit{checkNewActive}(s : \mathit{EState}, n : \mathit{LNode}) : \mathit{EState} & \\
\quad = \quad \textbf{if}\ s.\mathit{values}[n.\mathit{pNum}] \ = \ \texttt{true}\ \textbf{then} & \underline{\mathit{RNode\_signalRaises}}(s, n) \\
\qquad\quad \textbf{otherwise} & s
\end{array}
\tag{5.34}
$$

$$
\begin{array}{lll}
\boxed{newNode}\,(pred : LNode, o : \texttt{OrGr}, e : \mathcal{L}\,S') : RNode & & \\
\quad = \ \textbf{if } e = {}^{i,a}p_k \ \underline{;}\ \alpha \textbf{ then} & \texttt{Prime}\ (& expr = e, pNum = k \\
 & & predec = pred, livesIn = o \\
 & & endReached = \texttt{false} \\
 & & eFirst = eLast = \infty \\
 & & seFirst = \bot \\
 & & seLast = \\
 & & \quad \textbf{if } isLeading(\texttt{this}) \textbf{then} \quad \infty \\
 & & \quad \textbf{otherwise} \qquad\qquad pred.seLast \\
 & & tiPending = toPending = \infty \\
 & & sumMinPreds = calcMinPreds(\texttt{this}) \\
 & & sumMaxPreds = calcMaxPreds(\texttt{this}) \\
 & &) \\
\quad \textbf{if } e = \texttt{AND}_k \ \underline{;}\ \alpha \textbf{ then} & \texttt{ATst}\ (& expr = e, pNum = k \\
 & & predec = pred, livesIn = o \\
 & & sumMinPreds = calcMinPreds(\texttt{this}) \\
 & & sumMaxPreds = calcMaxPreds(\texttt{this}) \\
 & & seFirstCache = seLastCache = \infty \\
 & &)
\end{array}
\tag{5.35}
$$

$$
\begin{array}{l}
calcMinPreds(n : Node) : \mathbb{D} \\
\quad = \ \textbf{if } \underline{isLeading}(n) \textbf{then} \quad 0.0 \\
\qquad \textbf{otherwise} \qquad n.predec.sumMinPreds + \underline{minDura}(n.predec) \\
calcMaxPreds(n : Node) : \mathbb{D}_+ \\
\quad = \ \textbf{if } \underline{isLeading}(n) \textbf{then} \quad 0.0 \\
\qquad \textbf{otherwise} \qquad n.predec.sumMaxPreds + maxDura(n.predec)
\end{array}
\tag{5.36}
$$

$$
\begin{array}{lll}
\boxed{maxDura}\,(n : \texttt{Prime}) : & \mathbb{D}_+ = & a \textbf{ where } n.expr = {}^{\neg,a}p_{\_} \ \underline{;}\ e \\
\boxed{maxDura}\,(n : \texttt{ASol}) : & \mathbb{D}_+ = & n.maxSubSum
\end{array}
\tag{5.37}
$$

$$
\begin{array}{l}
\boxed{ATst\_installParts}\,(s : EState, a : \texttt{ATst}) : EState \\
\quad = \ \textbf{reduce}(O, installOrGroup(\_, a, \_), s) \\
\textbf{where } a.expr = \texttt{AND}(O)
\end{array}
\tag{5.38}
$$

$$
\begin{array}{l}
installOrGr(s : EState, a : \texttt{ATst}, e : \mathcal{L}\,t\_or) : EState \\
\quad = \ \textbf{reduce}(T, \underline{termInst}\,(\_, a.predec, o, (\_\ \underline{;}\ \triangle_{\triangle})), s) \\
\textbf{where} \quad o = \textbf{new } \texttt{OrGr}(tstPartOf = a, exprX = t) \\
\qquad\quad t = \texttt{OR}(T)
\end{array}
\tag{5.39}
$$

5.8 Creating Solutions of Conjunctions

$$
\begin{array}{l}
\boxed{LNode\_endReached}\,(s : EState, n : LNode) : EState \\
\quad = \; ATst\_newSolutions(s_1, n_1, n.livesIn, n.livesIn.tstPartOf) \\
\quad\; \textbf{where } n_1 = n \oplus (endReached = \texttt{true}) \\
\qquad\qquad\; s_1 = s \oplus (nodes \diamond n_1 \div n)
\end{array}
\tag{5.40}
$$

$$
\begin{array}{l}
ATst\_newSolutions(s : EState, n : LNode, o : \texttt{OrGr}, A : \texttt{ATst}) : EState \\
\quad = \; \textbf{reduce}\,(allCombinations(s.nodes, a, o), installSolution(\_, n, \_, A), s)
\end{array}
\tag{5.41}
$$

$$
\begin{array}{l}
installSolution(s : EState, n : LNode, N : \textbf{set of}\, LNode, A : \texttt{ATst}) : EState \\
\quad = \; \textbf{if}\, \neg\,(a.aeFirst \leq a.aeLast \;\wedge\; a.minSubSum \leq a.maxSubSum)\; \textbf{then}\; s \\
\qquad\; \textbf{otherwise} \\
\qquad\; \textbf{if}\; a.tiPending = \infty\; \textbf{then}\; \underline{LNode\_becomesValid}(s', a) \\
\qquad\; \textbf{otherwise}\; s' \\
\;\; \textbf{where}\;\; s' = s \oplus (nodes = nodes \cup \{a\}) \\
\qquad\quad a = \texttt{ASol}(\; solParts = N \cup \{n\} \\
\qquad\qquad aeFirst = \mathsf{latest}\,(\textbf{map}(solParts, \_.seFirst)) \\
\qquad\qquad aeLast = \mathsf{earliest}\,(\textbf{map}(solParts, \_.seLast)) \\
\qquad\qquad minSubSum = \mathsf{max}\,(\textbf{map}(solParts, minSum(\_))) \\
\qquad\qquad maxSubSum = \mathsf{min}\,(\textbf{map}(solParts, maxSum(\_))) \\
\qquad\qquad expr = \bot \\
\qquad\qquad predec = A.predec \\
\qquad\qquad livesIn = A.livesIn \\
\qquad\qquad endReached = \texttt{false} \\
\qquad\qquad sumMinPreds = A.sumMinPreds \\
\qquad\qquad sumMaxPreds = A.sumMaxPreds \\
\qquad\qquad tiPending = \begin{cases} aeFirst + minSubSum \\ \qquad \textbf{if}\; aeFirst + minSubSum > s.now \\ \infty \quad \textbf{otherwise} \end{cases} \\
\qquad\qquad toPending = \begin{cases} aeLast + maxSubSum \\ \qquad \textbf{if}\; aeLast + maxSubSum < \infty \\ \infty \quad \textbf{otherwise} \end{cases} \\
\qquad\qquad seFirst = \begin{cases} aeFirst \quad \textbf{if}\; \underline{isLeading}(a) \\ \mathsf{latest}\,\{predSeFirst, aeFirst - sumMaxPreds\} \\ \qquad\qquad \textbf{otherwise} \end{cases} \\
\qquad\qquad seLast = \begin{cases} aeLast \quad \textbf{if}\; \underline{isLeading}(a) \\ \mathsf{earliest}\,\{predSeLast, aeLast - sumMinPreds\} \\ \qquad\qquad \textbf{otherwise} \end{cases} \\
\qquad\qquad) \\
\qquad minSum(n : LNode) : \mathbb{T} \;=\; n.sumMinPreds + minDura(n) \\
\qquad maxSum(n : LNode) : \mathbb{T} \;=\; n.sumMaxPreds + maxDura(n) \\
\qquad predSeLast \;=\; \begin{array}{ll} pred.seLast & \textbf{if}\, A.seLastCache = \infty \\ A.seLastCache & \textbf{otherwise} \end{array} \\
\qquad predSeFirst \;=\; \begin{array}{ll} pred.seFirst & \textbf{if}\, A.seFirstCache = \infty \\ A.seFirstCache & \textbf{otherwise} \end{array}
\end{array}
\tag{5.42}
$$

$$
\begin{aligned}
&\mathit{allCombinations}(N : \mathit{ObjSet}, a : \texttt{ATst}, o : \texttt{OrGr}) : \textbf{set of set of } \mathit{LNode} \\
&\quad = \textbf{reduce}\,(\underline{\mathit{tstParts}}\,(N, a) \setminus \{o\}, \mathit{comb0}(N, \_, \_), \{\{\}\}) \\
&\mathit{comb0}(N : \mathit{ObjSet}, o : \texttt{OrGr}, \mathit{grown} : \textbf{set of set of } \mathit{LNode}) \\
&\quad = \textbf{map}\,(\mathit{finalNodes}(N, o), \mathit{comb1}(\_, \mathit{grown})) \\
&\mathit{comb1}(n : \mathit{LNode}, \mathit{grown} : \textbf{set of set of } \mathit{LNode}) \\
&\quad = \textbf{map}\,(\mathit{grown}, (\{n\} \cup \_))
\end{aligned}
\tag{5.43}
$$

This correct, but complicated definition can be explained by the following informal definition:

$$
\begin{aligned}
&\mathit{allCombinations}(N : \mathit{ObjSet}, a : \texttt{ATst}, o : \texttt{OrGr}) : \textbf{set of set of } \mathit{LNode} \\
&\quad = \mathit{convertTupleToSet}\ (\!|\ \mathit{finalNodes}(o_1) \times \ldots \times \mathit{finalNodes}(o_k)\ |\!) \\
&\quad \textbf{where } \{o_1, \ldots, o_k\} \ = \underline{\mathit{tstParts}}(N, a) \setminus \{o\}
\end{aligned}
\tag{5.44}
$$

5.9 Termination of Nodes

$$
\begin{array}{lll}
N \bigstar n_2 \div n_1 & : & \mathit{ObjSet} \times \texttt{Prime} \times \texttt{Prime} \rightarrow \mathit{ObjSet} \\
 & \cup & \mathit{ObjSet} \times \texttt{ASol} \times \texttt{ASol} \rightarrow \mathit{ObjSet} \\
N \bigstar n_2 \div n_1 & = & N_2 \\
\textbf{where} \quad N_1 & = & \textbf{reduce}\,(\mathit{successors}(n_1), \\
 & & \quad (\lambda\, X, x \bullet X \bigstar (x \oplus (\mathit{predec} = n_2)) \div x\ ), N) \\
\quad\quad N_2 & = & \textbf{reduce}\,(\mathit{usedInSolutions}(n_1), \\
 & & \quad (\lambda\, X, x \bullet X \bigstar (x \oplus (\mathit{solParts} = \mathit{solParts} \setminus \{n_1\} \cup \{n_2\})) \div x\ ), N_1)
\end{array}
\tag{5.45}
$$

$$
\begin{array}{l}
\boxed{\mathit{Prime\_terminates}}\,(N : \mathit{ObjSet}, \mathit{now} : \mathbb{T}, n : \texttt{Prime}) : \mathit{ObjSet} \\
\quad = \quad \textbf{if}\, \neg\ \mathit{isFixed}(n) \quad \mathit{LNode\_terminates}(N \cup \underline{\mathit{newnode}}(n.\mathit{predec}, n.\mathit{livesIn}, n.\mathit{expr}), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{now}, n) \\
\qquad \textbf{otherwise} \qquad\quad \mathit{LNode\_terminates}(N, \mathit{now}, n)
\end{array}
\tag{5.46}
$$

$$
\begin{array}{lll}
\boxed{\mathit{LNode\_terminates}}\,(N : \mathit{ObjSet}, \mathit{now} : \mathbb{T}, n : \mathit{LNode}) : \mathit{ObjSet} & = & N_3 \\
\textbf{where} \quad N_1 & = & \textbf{reduce}\ (\mathit{successors}(N, n), \mathit{RNode\_becomesFixed}(\_, \_), N) \\
\quad\quad N_2 & = & \textbf{reduce}\ (\mathit{usedInSolutions}(N_1, n), \mathit{LNode\_terminates}(\_, t, \_), N_1) \\
\quad\quad N_3 & = & \mathit{ATst\_alternativeGetsLost}(N_2, n)\ \boxed{\setminus \{n\}}
\end{array}
\tag{5.47}
$$

$$
\begin{array}{l}
\mathit{isFixed}(n : \texttt{Prime}) : \mathit{Boolean} \ = \ (n.\mathit{eLast} \neq \infty) \\
\mathit{successors}(N : \mathit{ObjSet}, n : \mathit{LNode}) : \ \textbf{set of}\, \mathit{RNode} \\
\qquad\qquad\qquad\qquad\qquad\qquad = \ \{x \in N \mid x.\mathit{predec} = n\} \\
\mathit{usedInSolutions}(N : \mathit{ObjSet}, n : \mathit{LNode}) : \ \textbf{set of}\, \texttt{ASol} \\
\qquad\qquad\qquad\qquad\qquad\qquad = \ \{x \in N \cap \texttt{ASol} \mid n \in x.\mathit{solParts}\}
\end{array}
\tag{5.48}
$$

$$
\begin{array}{l}
RNode\_becomesFixed(N : ObjSet, now : \mathbb{T}, n : \texttt{Prime}) : ObjSet \\
\quad = \quad \textbf{if}\ \underline{isTesting}(n)\ \textbf{then}\quad (ATst\_alternativeGetsLost(N, n))\ \boxed{\setminus \{n\}} \\
\qquad\quad \textbf{otherwise} \qquad\qquad LNode\_SEL\_lowers(N \bigstar n' \div n, newSEL, now, n') \\
\quad \textbf{where}\ newSEL = now - n.sumMinPreds \\
\qquad\quad n' = n \oplus (toPending = now + \underline{maxDura}(n), eLast = now)
\end{array}
\tag{5.49}
$$

$$
\begin{array}{l}
RNode\_becomesFixed(N : ObjSet, now : \mathbb{T}, n : \texttt{ATst}) : ObjSet \\
\quad = \quad \textbf{if}\ \underline{isTesting}(n)\ \textbf{then}\quad (ATst\_alternativeGetsLost(N, n))\ \boxed{\setminus \{n\}} \\
\qquad\quad \textbf{otherwise} \qquad\qquad N \bigstar n' \div n \\
\quad \textbf{where}\ n' = n \oplus (\ seFirstCache = n.pred.seFirst; \\
\qquad\qquad\qquad seLastCache = \textbf{earliest}(n.pred.seLast, now - n.sumMinPreds)
\end{array}
\tag{5.50}
$$

$$
\begin{array}{l}
LNode\_SEL\_lowers(N : ObjSet, t : \mathbb{T}, now : \mathbb{T}, n : LNode) : ObjSet \\
\quad = \quad \textbf{if}\ t \geq n.selLast\ \textbf{then}\quad N \\
\qquad\quad \textbf{otherwise} \qquad\qquad N_2 \bigstar n \oplus (seLast = t) \div n \\
\quad \textbf{where}\ S = \{m \mid m \in successors(N, n)\ \wedge\ \neg\ isLeading(m)\} \\
\qquad\quad N_1 = \textbf{reduce}(S, LNode\_SEL\_lowers(\_, t, now, \_), N) \\
\qquad\quad N_2 = \textbf{reduce}(usedInSolutions(N, n), ASol\_subSEL\_lowers(\_, t, now, \_), N_1)
\end{array}
\tag{5.51}
$$

$$
\begin{array}{l}
ASol\_subSEL\_lowers(N : ObjSet, t : \mathbb{T}, now : \mathbb{T}, n : \texttt{ASol}) : ObjSet \\
\quad = \quad \textbf{if}\ t < n.aeFirst\quad \textbf{then}\ ATst\_alternativeGetsLost(N, n)\ \boxed{\setminus \{n\}} \\
\qquad\quad \textbf{if}\ t \geq n.aeLast\quad \textbf{then}\ N \\
\qquad\quad \textbf{otherwise} \\
\qquad\quad \textbf{if}\ n_1.toPending > now\ \textbf{then} \\
\qquad\qquad\quad LNode\_SEL\_lowers(N_1, t - sumMinPreds(n), now, n_1) \\
\qquad\quad \textbf{otherwise} \qquad LNode\_terminates(N_1, now, n_1) \\
\quad \textbf{where}\ n_1 = n \oplus (aeLast = t;\ toPending = t + maxSubSum) \\
\qquad\quad N_1 = N \bigstar n_1 \div n
\end{array}
\tag{5.52}
$$

$$
\begin{array}{l}
ATst\_alternativeGetsLost(N : ObjSet, n : Node) : ObjSet \\
\quad = \quad \textbf{if}\ \underline{allInhabitants}(N, n.livesIn) \setminus \{n\} = \{\}\ \textbf{then}\quad N_2 \\
\qquad\quad \textbf{otherwise} \qquad\qquad\qquad\qquad\qquad\qquad N \\
\quad \textbf{where}\quad N_1 = deleteAll(N, n.livesIn.tstPartOf) \\
\qquad\qquad\quad N_2 = ATst\_alternativeGetsLost(N_1, n.livesIn.tstPartOf)
\end{array}
\tag{5.53}
$$

$$
\begin{array}{l}
deleteAll(N : ObjSet, n : \texttt{ATst}) : ObjSet \quad = \\
\quad (\textbf{reduce}(\underline{tstParts}(N, n), deleteAll(\_, \_), N))\ \boxed{\setminus \{n\}} \\
deleteAll(N : ObjSet, n : \texttt{OrGr}) : ObjSet \quad = \\
\quad (\textbf{reduce}(\underline{allInhabitants}(N, n), deleteAll(\_, \_), N))\ \boxed{\setminus \{n\}} \\
deleteAll(N : ObjSet, n : \texttt{Prime}) : ObjSet \quad = \quad N\ \boxed{\setminus \{n\}} \\
deleteAll(N : ObjSet, n : \texttt{ASol}) : ObjSet \quad = \quad N\ \boxed{\setminus \{n\}}
\end{array}
\tag{5.54}
$$

Chapter 6

Proofs

6.1 Structure of this Chapter

This chapter contains proofs w.r.t. several properties of the algorithm.

The most important of these propoerties are called *correctness* and *completeness* in the following.

Correctness means, that if the algorithm delivers a `pass` verdict, the SUT's trace indeed fulfills SpecUT.

Completeness means, that if the SUT's trace fulfills SpecUT, then the algorithm delivers a `pass` verdict.

Further proof obligations concern the *termination* of the algorithm, and the correctness of the way in which the algorithm treats nodes in the terminated state. These proofs are distributed to the sections of this chapter as follows :

- Section 6.3 demonstrates the fundamental lemma for the proof of correctness, which says that each valid node object represents an interpretation of a prefix of the SUT's behaviour w.r.t. a prefix of one linear specification derived from SpecUT.
- Section 6.4 demonstrates the fundamental lemma for the proof of completeness, which says that each partial interpretation of the SUT's behaviour implies the existence of a valid node object.
- Section 6.5 applies the results of section 6.3 and 6.4 to show the correctness and completeness of the final and early verdicts.
- Section 6.6 demonstrates that the execution of the algorithm always terminates.
- Section 6.7 demonstrates that a specific optimization applied in the implementation, namely the complete deletion of node objects which have reached the terminated state, does not affect the other properties.

The proofs concerning correctness and the treatment of terminated node objects are widely formalized, — w.r.t. the other targets this seems neither necessary nor useful.

In spite of the non-deterministically defined functions **map**() and **reduce**() being frequently used in the definition of the algorithm in chapter 5, no explicit proof of *confluence* is necessary : Since the algorithm only delivers one of two values, confluence is implied by the conjunction of correctness and completeness.

6.2 Notational Conventions and Global Abbreviations

In the following text, let ...

$$D = D_{[t_{startSession} \ldots t_{endSession}]} \in \mathcal{R}_+$$

be a certain SUT's trace data during a complete test session.

Further, $D_{[t_1 \ldots t_2]} \in \mathcal{R}_+$ with $t_{startSession} \leq t_1 < t_2 \leq t_{endSession}$ denotates a non-empty sub-trace of D extending from t_1 to t_2.

Further, in the following text the notation ...

$$v_k[t] \tag{6.1}$$

refers to the value of the observation function indexed by the atomic predicate p_k at the time instant t.

Note that t must not be a critical time instant w.r.t. p_k, because at these time instants a single value for v_k cannot be given.[1]

Further we define a function *expHd*, which delivers the own specification particle of a `Prime` node, which is the head of the subsequent expression for which the node has been created (cf. section 4.7.5) :

$$\begin{aligned} &expHd \ : \ \texttt{Prime} \rightarrow \mathcal{L}\,S' \\ &(n.expr \ = \ {}^{i,a}p_k \ \underline{;} \ \beta) \iff (n.expHd \ = \ {}^{i,a}p_k) \end{aligned} \tag{6.2}$$

Then we can calculate the linear specification, for which a given node represents partial interpretations, by ...

$$\begin{aligned} &\mathsf{SPath}\ \_ \ : \ LNode \rightarrow \mathcal{L}\,S'' \\ &\mathsf{SPath}\ n \ = \ \begin{cases} n.expHd & \textbf{if } n.predec = n_{-1} \\ (\mathsf{SPath}\ (n.predec)) \ \underline{;} \ n.expHd & \textbf{otherwise} \end{cases} \end{aligned} \tag{6.3}$$

...and a function for extracting sub-sequences of this linear interpretation, which extend from one node to one of its transitive successor nodes, by ...

$$\begin{aligned} &\mathsf{SPath}\ \_ \rightsquigarrow \_ \ : \ LNode \times LNode \rightarrow \mathcal{L}\,S'' \\ &\mathsf{SPath}\ n_f \rightsquigarrow n_t \ = \ \begin{cases} n_f.expHd & \textbf{if } n_f = n_t \\ (\mathsf{SPath}\ n_f \rightsquigarrow n_t.predec) \ \underline{;} \ n_t.expHd & \textbf{otherwise} \end{cases} \end{aligned} \tag{6.4}$$

[1] This imposes no severe problem, because the same "trick" can be used as mentioned in the footnote on page 23.

The following formulæ and considerations refer to the objects and their attributes of the evaluating machine, as described in the previous chapter. In the formulæ contained therein, the transformations are defined which are applied to the data state of the algorithm in *one single* evaluation step. Their definitions have been given as pure and time-less functions.

Contrarily, the following interpretations have to consider the evolution of the data state in course of the test session, i.e. an ordered sequence of *multiple* data states, caused by the execution of multiple evaluation steps.

In the physical reality of the implementation of course *all* data values and object states are functions from time into some range. But most of these time dependencies are restricted just to the creation and initialization of an object, and most of the data values do take a constant value during the whole further development. For the sake of readability, these trivial time dependencies will be left *implicit*.

Contrarily, whenever an attribute of an object is intended to change during execution, it is *explicitly* modeled as a function from $\mathbb{T}$ into its range, and all references to such an attribute are written as function applications. The domain value, which is a time instant, is written as an *index*. By this notation a maximal similarity is achieved between the graphical appearances of the time-less formulæ in chapter 5 and the corresponding dynamic formulæ in this chapter.

Let n be a node object, and g be the global state the algorithm works on, then we will write in the following text :

$n.eFirst$	...because $.eFirst$ will stay constant throughout the life-time of the node object n.
$n.predec$	...because $.predec$ is fixed with the creation of the node object.
$n_{t_{now}}.elast$	...because the value of $.eLast$ may change with each evaluation step.
$isValid(n_{t_0})$	...because the function result depends on non-constant attribute values.
$finalNodes(g_t.top)$	...because the global state is dynamic.

(6.5)

Note that e.g. the notations $finalNodes(g_t.top)$, $finalNodes(g.top_t)$ and even $finalNodes_t(g.top)$ are equivalent, since the certain attribute value itself as well as the object as a whole can be regarded as a dynamically defined value, — the parameter modeling the "current time" can be inserted at any position in the sequence of parameters, because it is uniquely identified by this notation.

According to the structure of the inductive derivations contained in this chapter, the following notation is used:

$$
\begin{array}{lll}
\alpha & & \\
\wedge & \beta & \\
 & \hline \beta' & \\
 & \hline \beta'' & \\
 & \hline \cdots & \\
\gamma & & \\
\hline \gamma' & & \\
\hline \cdots & &
\end{array}
$$

It denotates graphically (1) that α and β are some premises, (2) that $\beta', \beta'', \ldots$ is a chain of conclusions the first of which is derived only from β, and (3) that γ is drawn from α and the last $\beta''^{\cdots}$.

6.3 Correctness

6.3.1 Contents and Structure of this Section

The derivations in this chapter prove that each currently valid node represents an interpretation of the prefix of the SUT's traces w.r.t. a prefix of one linearization of SpecUT. This is expressed by the central lemma (6.27) on page 64.

This lemma is derived in an inductive way by the following steps:

- Section 6.3.2 demonstrates that each valid `Prime` node represents the maximal set of segments, which fulfill its specification particle and start when the node's predecessor has been valid.
- Section 6.3.3 demonstrates by simple induction, that each valid `Prime` node represents a partial interpretation, if SpecUT is a simple chop sequence of ${}^{i,a}p_k$ specifications.
- Section 6.3.4 derives the semantics of nodes which are part of a node chain corresponding to a sub-expression of an `AND/OR` expression.
- Section 6.3.5 constructs the semantics of `ASol` nodes, using the result of the preceding section.
- Section 6.3.6 demonstrates that `ASol` nodes which correspond to `AND/OR` expressions containing only `Prime` nodes, can be embedded into the top level specification expression without losing the central result of section 6.3.4.
- Section 6.3.7 proves that the results of section 6.3.4 still hold when `ASol` nodes are embedded into these sub-expressions of `AND/OR` expressions, which makes the central lemma (6.27) valid for arbitrarily nested specification terms.

6.3.2 States and Local Semantics of Prime Nodes

The semantics properties of node objects are based on the definition of a *valid node*.

For ASol nodes a definition is given in (6.46) on page 73.

In case of *Prime* nodes[2] the different states under discussion (cf. figure 4.3) are defined by ...

$$\begin{array}{lcl} isTesting(n : Prime) & \Longleftrightarrow & (n.eFirst = \infty) \\ timeIn(n : Prime) & \Longleftrightarrow & (n.eFirst \neq \infty \wedge n.tiPending \neq \infty) \\ isValid(n : Prime) & \Longleftrightarrow & (n.eFirst \neq \infty \wedge n.tiPending = \infty) \end{array} \tag{6.6}$$

The fundamental property of each valid *Prime* node is given by the following lemma (6.7). A corresponding property will be defined later also for ASol-nodes, see formula (6.47) in section 6.3.5 on page 74, so that all *LNode* objects can be treated in a uniform way.

$$\begin{array}{lll} \forall\ t_{now}, t : \mathbb{T},\ n : Prime & | & isValid\ n_{t_{now}} \\ & \wedge & n.expHd = {}^{i,a}p_k \\ & \wedge & t_0 = \mathsf{latest}(n.eFirst, t_{now} - a) \\ & \wedge & t_2 = \mathsf{earliest}(n_{t_{now}}.eLast, t_{now} - i) \\ \bullet\ t_0 \leq t \leq t_2 & \Longleftrightarrow & \left(D_{[t \ldots t_{now}]} \in [[\,{}^{i,a}p_k\,]]^L \wedge isValid\ (n.predec)_t \right) \\ \wedge\ t_0 \leq t_2 & & \end{array} \tag{6.7}$$

This property means that for a given *Prime* node n, which is in a *valid* state at some time instant t_{now}, the interval from t_0 to t_2 is the maximal set of real-time instants which can be used as a segment's start time, if this segment shall (1) fulfill the specification particle ${}^{i,a}p_k$, (2) begin at some time instant when the node serving as $n.predec$ was in a *valid* state, and (3) extend up to t_{now}.

It secondly implies that this set of intervals is always non-empty.

The first consequence of formula (6.7) can be decomposed into a conjunction of three propositions, the validity of which is demonstrated separately as follows :

$$\begin{array}{l} (\ t_0 \leq t \leq t_2 \Longleftrightarrow \left(D_{[t \ldots t_{now}]} \in [[\,{}^{i,a}p_k\,]]^L \wedge isValid\ (n.predec)_t \right)) \\ = (\quad t_0 \leq t \leq t_2 \Longrightarrow \left(D_{[t \ldots t_{now}]} \in [[\,{}^{i,a}p_k\,]]^L \wedge isValid\ (n.predec)_t \right) \\ \quad \wedge\ t < t_0 \Longrightarrow \left(\neg\ D_{[t \ldots t_{now}]} \in [[\,{}^{i,a}p_k\,]]^L \right) \vee (\neg\ isValid\ (n.predec)_t\ ) \\ \quad \wedge\ t_2 < t \Longrightarrow \left(\neg\ D_{[t \ldots t_{now}]} \in [[\,{}^{i,a}p_k\,]]^L \right) \vee (\neg\ isValid\ (n.predec)_t\ ) \\ \quad) \end{array} \tag{6.8}$$

[2] For the sake of readability, the notation "*Prime*" is used in the following text as equivalent to the notation "Prime".

Any node n which is in a valid state at time instant t_{now} can have reached this state in one of three ways (cf. figure 4.3):

(1) The node n has been created in the testing state at some time instant t_p, in the same evaluation step in which its predecessor $n.predec$ has become valid. This is implemented by the procedure $nextInst()^{(5.32)}$ being immediately called by $LNode\_becomesActive()^{(5.31)}$. Since further each node can become valid only once in its life cycle, it further holds that

$$\forall\, t : \mathbb{T} \mid t \leq t_p \bullet \neg\, isValid(n.predec)_t \tag{6.9}$$

The node n changes to the time-in state immediately at t_p, in the same evaluation step, iff v_k has already been `true` and stays `true`, or changes to `true` at this very moment (both expressed by $v_k[t_p] = \texttt{true}$).

This is implemented by the procedure $checkNewActive()^{(5.34)}$, which is called by $termInst()^{(5.33)}$ after n has been created in the testing state. In this case it holds that $t_p = n.eFirst$.

(2) Otherwise n changes to the time-in state at some later time instant, namely as soon as v_k becomes `true` at $n.eFirst > t_p$. This can happen only in the positive part of a subsequent evaluation step, performed by the function $RNode\_signalRaises()^{(5.20)}$, which is called on every node which is in a testing state when the corresponding external predicate becomes `true` in this very evaluation step.

So in both cases following negative consequences hold:

$$\begin{array}{lll} \forall\, t_{now}, t : \mathbb{T},\ n : Prime & \mid & isValid\ n_{t_{now}} \\ & \wedge & n.expHd = {}^{i,a}p_k \\ \bullet\quad t < t_p & \Longrightarrow & \neg\ isValid(n.predec)_t \\ \wedge\quad t_p \leq t < n.eFirst & \Longrightarrow & \neg\ v_k[t] = \texttt{true} \end{array} \tag{6.10}$$

...which can be rewritten by applying the definition of $[[\texttt{--}p_k]]^L$ to ...

$$\begin{array}{lll} \ldots & & \\ \bullet\quad t < t_p & \Longrightarrow & \neg\ isValid(n.predec)_t \\ \wedge\quad t_p \leq t < n.eFirst & \Longrightarrow & D_{[t \ldots t_{now}]} \notin [[\texttt{--}p_k]]^L \end{array} \tag{6.11}$$

(3) In a third case n is created in the testing state, as soon as a different node n', with identical values of $.predec$ and $.expr$ terminates at the time instant t_q, while $n.predec$ is still valid. This is realized by calling $newNode()^{(5.35)}$ in course of the execution of $Prime\_terminates()^{(5.46)}$, and is needed for catching a subsequent becoming-true-again of the observation function v_k for recognizing the different possible interpretations of the input data, cf. the last two node objects representing both "$p\ \underline{;}\ q\ \underline{;}\ r$" in figure 4.2.

Since v_k cannot become externally `false` and `true` in the same evaluation step, the node n can enter a valid state not before a non-zero duration has passed after t_q. Therefore, in this third case it holds that ...

$$\begin{array}{rl} \forall\ t_{now}, t : \mathbb{T},\ n : Prime & \mid\ isValid\ n_{t_{now}} \\ & \wedge\ n.expHd = {}^{i,a}p_k \\ \bullet\quad t_p < t_q < t < n.eFirst & \Longrightarrow\ \neg\ v_k[t] = \texttt{true} \end{array} \tag{6.12}$$

...which can again be rewritten by applying the definition of $[[\texttt{--}p_k]]^L$ to ...

$$\begin{array}{l} \ldots \\ \bullet\quad t < n.eFirst \Longrightarrow D_{[t\ldots t_{now}]} \notin [[\texttt{--}p_k]]^L \end{array} \tag{6.13}$$

From formula (6.11) as well as from (6.13) it follows that ...

$$\begin{array}{rl} \forall\ t_{now}, t : \mathbb{T},\ n : Prime & \mid\ isValid\ n_{t_{now}} \\ & \wedge\ n.expHd = {}^{i,a}p_k \\ \bullet\quad t < n.eFirst \Longrightarrow & \neg\ isValid(n.predec)_t \\ & \vee\ D_{[t\ldots t_{now}]} \notin [[\texttt{--}p_k]]^L \end{array} \tag{6.14}$$

In all three cases there had been no intervening evaluation step between $n.eFirst$ and t_{now} in the negative phase of which n left its valid state, because this would imply the transition of n into the terminated state, from which no valid state is reachable, — a contradiction to $isValid(n_{t_{now}})$.

Since a transition to the terminated state must occurs when the observation function corresponding to a valid Prime node goes `false` (5.16)(5.19)(5.47), it holds that

$$\begin{array}{rl} \forall\ t_{now}, t : \mathbb{T},\ n : Prime & \mid\ isValid\ n_{t_{now}} \\ & \wedge\ n.expHd = {}^{i,a}p_k \\ \bullet\quad n.eFirst \leq t \leq t_{now} & \Longrightarrow\ v_k[t] = \texttt{true} \end{array} \tag{6.15}$$

...from which follows a fundamental semantic property, namely that all segments starting at any time instant $\geq n.eFirst$ fulfill ${}^{0,\infty}p_k$, which is the specification particle of n without its duration requirements:

$$\begin{array}{rl} \forall\ t_{now}, t : \mathbb{T},\ n : Prime & \mid\ isValid\ n_{t_{now}} \\ & \wedge\ n.expHd = {}^{i,a}p_k \\ \bullet\quad n.eFirst \leq t \leq t_{now} & \Longrightarrow\ D_{[t\ldots t_{now}]} \in [[{}^{0,\infty}p_k]]^L \end{array} \tag{6.16}$$

In the evaluation step when n entered the time-in state, the node $n.predec$ has been valid, too. This is because the leaving of its valid state is signaled to n by calling *RNode_becomesFixed()*, which would have removed the node n from the object set in case it would still have been in the testing state (5.49).

If n is in the time-in or active state, the time instant of this event is recorded in $n.eLast$, which changes from ∞ to the current time. So it follows that ...

$$\begin{array}{rl} \forall\ t_{now}, t : \mathbb{T},\ n : Prime & \mid\ isValid\ n_{t_{now}} \\ \bullet\quad n.eFirst \leq t \leq \mathsf{earliest}(t_{now}, n_{t_{now}}.eLast) & \Longrightarrow\ isValid(n.predec)_t \\ \wedge\quad n_{t_{now}}.eLast < t & \Longrightarrow\ \neg\ isValid(n.predec)_t \end{array} \tag{6.17}$$

If the expressions $n.expHd$ has a minimum duration requirement $i > 0.0$, all sub-traces which begin at t and shall end at time instant t_{now} must fulfill the condition $t_{now} - t \geq i$, according to the definition of the semantics of $\texttt{MIN}\, i\, p$ in formula (3.3) above. Rewriting to $t \leq t_{now} - i$ shows, that these sub-traces must not begin later than $t_{now} - i$. Because a valid state is entered for any node at time instant $eFirst + i$, when the min-timer expires[5.21], it is always guaranteed for a valid node that $t_{now} \geq eFirst + i$.

So we get ...

$$\begin{array}{lll} \forall\, t_{now}, t : \mathbb{T},\ n : Prime & \mid & isValid\ n_{t_{now}} \\ & \wedge & n.expHd = {}^{i,a}p_k \\ \bullet\ \ n.eFirst \leq t \leq t_{now} - i & \Longrightarrow & D_{[t \ldots t_{now}]} \in [[{}^{i,\infty}p_k]]^L \\ \wedge\ \ t_{now} - i < t & \Longrightarrow & D_{[t \ldots t_{now}]} \notin [[{}^{i,\infty}p_k]]^L \end{array} \tag{6.18}$$

Similar, if a maximal duration requirement $a < \infty$ is imposed on ${}^{\neg,a}p_k$, a sub-trace $D_{[t \ldots t_{now}]} \in [[{}^{\neg a}p_k]]^L$ (i.e. a sub-trace which fulfills ${}^{\neg,a}p_k$ and extends up to t_{now}) must fulfill the condition $t_{now} - t \leq a$, and begin not earlier than $t_{now} - a$.

So we get ...

$$\begin{array}{lll} \forall\, t_{now}, t : \mathbb{T},\ n : Prime & \mid & isValid\ n_{t_{now}} \\ & \wedge & n.expHd = {}^{i,a}p_k \\ \bullet\ \ \mathsf{latest}(n.eFirst, t_{now} - a) \leq t \leq t_{now} & \Longrightarrow & D_{[t \ldots t_{now}]} \in [[{}^{0.0,a}p_k]]^L \\ \wedge\ \ t < t_{now} - a & \Longrightarrow & D_{[t \ldots t_{now}]} \notin [[{}^{0.0,a}p_k]]^L \end{array} \tag{6.19}$$

Now the different inequalities with negated consequences can be disjunctively combined as follows:

$$\begin{array}{llllll} \forall\, t_{now}, t : \mathbb{T},\ n : Prime & \mid & isValid\ n_{t_{now}} \wedge n.expHd = {}^{i,a}p_k & \bullet & & \\ t < n.eFirst & \Longrightarrow & \neg\ isValid(n.predec)_t & \vee & D_{[t \ldots t_{now}]} \notin [[{}^{\neg\neg}p_k]]^L & (6.14) \\ t < t_{now} - a & \Longrightarrow & & & D_{[t \ldots t_{now}]} \notin [[{}^{0,a}p_k]]^L & (6.19) \\ \hline t < \mathsf{latest}(n.eFirst, t_{now} - a) & \Longrightarrow & \neg\ isValid(n.predec)_t & \vee & D_{[t \ldots t_{now}]} \notin [[{}^{i,a}p_k]]^L & \\ & & & & & \\ n_{t_{now}}.eLast < t & \Longrightarrow & \neg\ isValid(n.predec)_t & & & (6.17) \\ t_{now} - i < t & \Longrightarrow & & & D_{[t \ldots t_{now}]} \notin [[{}^{i,\infty}p_k]]^L & (6.18) \\ \hline \mathsf{earliest}(n_{t_{now}}.eLast, t_{now} - i) < t & \Longrightarrow & \neg\ isValid(n.predec)_t & \vee & D_{[t \ldots t_{now}]} \notin [[{}^{i,a}p_k]]^L & \end{array} \tag{6.20}$$

The inequalities with positive consequences must be conjugated:

$$\begin{array}{llllll} \forall\, t_{now}, t : \mathbb{T},\ n : Prime & \mid & isValid\ n_{t_{now}} \wedge n.expHd = {}^{i,a}p_k & \bullet & & \\ n.eFirst & \leq t \leq & \mathsf{earliest}(t_{now}, n_{t_{now}}.eLast) & \Longrightarrow & isValid(n.predec)_t & (6.17) \\ n.eFirst & \leq t \leq & t_{now} - i & \Longrightarrow & D_{[t \ldots t_{now}]} \in [[{}^{i,\infty}p_k]]^L & (6.18) \\ \mathsf{latest}(n.eFirst, t_{now} - a) & \leq t \leq & t_{now} & \Longrightarrow & D_{[t \ldots t_{now}]} \in [[{}^{0.0,a}p_k]]^L & (6.19) \\ \hline \mathsf{latest}(n.eFirst, t_{now} - a) & \leq t \leq & \mathsf{earliest}(t_{now} - i, n_{t_{now}}.eLast) & \Longrightarrow & isValid(n.predec)_t & \\ & & & & \wedge\ D_{[t \ldots t_{now}]} \in [[{}^{i,a}p_k]]^L & \end{array} \tag{6.21}$$

Since the conclusions of formulae (6.20) and (6.21) correspond to the three statements in (6.8), the first consequence of the fundamental property (6.7) has been demonstrated.

6.3.2.1 Proof of the Non-Emptiness of $\{t_0 \ldots t_2\}$

To demonstrate the non-emptiness of the interval in which (6.7) holds, means to show that ...

$$isValid(n_{t_{now}}) \implies \mathsf{latest}(n.eFirst, t_{now} - a) \leq \mathsf{earliest}(n_{t_{now}}.eLast, t_{now} - i) \quad (6.22)$$

This implication holds, because its consequence holds in all combinations of left and right sides:

$$n.eFirst \leq n_{t_{now}}.eLast \quad (6.23)$$

...because in the evaluation step when n entered the time-in state, the node $n.predec$ has been valid, — cf. the argumentation for formula (6.17) starting on page 61.

$$n.eFirst \leq t_{now} - i \quad (6.24)$$

...because the state transition from the time-in state into the valid state cannot happen earlier than that the corresponding min-timer expires at time instant $n.eFirst + i$.

$$t_{now} - a \leq n_{t_{now}}.eLast \quad (6.25)$$

...because either the node predecessor node is still valid at t_{now}, so that $n_{t_{now}}.elast = \infty$. Otherwise, if $n.predec$ has terminated, the proposition follows from the fact that a time-out request for $n_{t_{now}}.eLast + a$ has been installed $^{(5.47)(5.49)}$, which has obviously not yet expired, because n is still valid.

$$t_{now} - a \leq t_{now} - i \quad (6.26)$$

...because $a \geq i$, according to the definition of ${}^{i,a}p_k$, which is checked by the rewriting from S to S' statically, cf. the non-syntactic "**where**..." condition in table 4.1 and the informal description on page 18.

6.3.3 `Prime` Nodes Representing Partial Interpretations w.r.t. Top Level Chop Sequences

The lemma which is the final purpose of all demonstrations in this section 6.3 applies to all *LNodes* which correspond to specification particles from a top level chop sequence, i.e. from a linear specification which is directly derived from SpecUT, not from a sub-expression of an `AND/OR` expression.

This lemma is ...

$$\boxed{\begin{array}{l} \forall\ t_{now} : \mathbb{T},\ n : LNode \\ \quad \bullet\ isValid\ n_{t_{now}} \implies D_{[t_{startSession} \ldots t_{now}]} \in [[\,\mathsf{SPath}\ n\,]]^L \end{array}} \tag{6.27}$$

In the case of chop sequences containing `Prime` nodes only, i.e. in the case that SpecUT does not contain `AND/OR` expressions, its derivation is possible by induction, using the result of the preceding section:

(1)

The nodes corresponding to the first sub-expressions in every chop sequence are created in the testing state before the start of the test session by the procedure $iInit()^{(5.8)}$. For all these nodes the value of $.predec$ is the pseudo-node n_{-1}, representing the condition "test session not yet started".

The node n_{-1} is treated specially by the function $evalNodes\_initial()^{(5.15)}$, simulating a corresponding (pseudo-)observation function[3] which changes to `false` at time instant $t_{startSession}$.

So in this very first evaluation step only those nodes the corresponding observation function of which is `true` at $t_{startSession}$ can make the transition from the testing into the time-in state in the positive phase. All nodes which are still in the testing state in the negative phase, because the observation function has not been `true`, are deleted, because there predecessor terminates, cf. figure 4.3.

Therefore it holds for all of those nodes n which subsequently reach a valid state, that ...

$$\forall\, t_{now} : \mathbb{T} \bullet\ n.eFirst = n_{t_{now}}.eLast = t_{startSession} \tag{6.28}$$

[3] The definition of observation functions in section 2.1 restricts their domain to the interval of the test session, which is not the case with this theoretical "pseudo-" observation function.

W.r.t. the fundamental local node property of each Prime node (6.7) t_0 and t_2 calculate as follows:

$$\begin{array}{lcll} t_0 & = & \mathsf{latest}(n.eFirst, t_{now} - a) & \\ & = & \mathsf{latest}(n.eLast_{t_{now}}, t_{now} - a) & (6.28) \\ & = & n_{t_{now}}.eLast & (6.25) \\ & = & t_{startSession} & (6.28) \\ & & & \\ t_2 & = & \mathsf{earliest}(n_{t_{now}}.eLast, t_{now} - i) & \\ & = & \mathsf{earliest}(n.eFirst, t_{now} - i) & (6.28) \\ & = & t_{startSession} & (6.24) \end{array} \tag{6.29}$$

...so that for all immediate successors of n_{-1}, the leading nodes of top level chains, formula (6.7) takes the form ...

$$\begin{array}{l} \forall\ t_{now}, t : \mathbb{T},\ n : Prime \quad | \quad isValid\ n_{t_{now}} \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge \quad n.expHd = {}^{i,a}p_k \\ \bullet \quad t_{startSession} \leq t_{startSession} \\ \wedge \quad t_{startSession} \leq t \leq t_{startSession} \\ \qquad \Longleftrightarrow \ \left(D_{[t \ldots t_{now}]} \in [[\,{}^{i,a}p_k\,]]^L \wedge \ isValid\ (n.predec)_{t_{startSession}} \right) \end{array} \tag{6.30}$$

...which can be generalized and simplified to ...

$$\begin{array}{l} \forall\ t_{now}, t : \mathbb{T},\ n : Prime \ \mid \ n.predec = n_{-1} \ \wedge \ isValid\ n_{t_{now}} \ \wedge \ n.expHd = {}^{i,a}p_k \\ \bullet\ t = t_{startSession} \ \Longleftrightarrow \ D_{[t \ldots t_{now}]} \in [[\,{}^{i,a}p_k\,]]^L \end{array} \tag{6.31}$$

Since for each very first node of a specification expression it holds that $n.expHd = \mathsf{SPath}\ (n)$, the consequence can be written as ...

$$\begin{array}{l} \forall\ t_{now}, t : \mathbb{T},\ n : Prime \ \mid \ n.predec = n_{-1} \ \wedge \ isValid\ n_{t_{now}} \ \wedge \ n.expHd = {}^{i,a}p_k \\ \bullet\ t = t_{startSession} \ \Longleftrightarrow \ D_{[t \ldots t_{now}]} \in [[\,\mathsf{SPath}\ n\,]]^L \end{array} \tag{6.32}$$

(2)

Let it be assumed that for the predecessor $n.predec \neq n_{-1}$ of any node n it holds that ...

$$isValid(n.predec)_{t_{now}} \ \Longrightarrow \ D_{[t_{startSession} \ldots t_{now}]} \in [[\mathsf{SPath}\ (n.predec)]]^L \tag{6.33}$$

Then (6.27) can be derived from (6.7) and (6.32) by the following induction:

$$
\begin{array}{l}
\dfrac{\begin{array}{l}
(\ \forall\, t : \mathbb{T}, n : Prime \bullet isValid\ (n.predec)_t \implies D_{[t_{startSession} \ldots t]} \in [[\mathsf{SPath}\ \ n.predec]]^L) \\
\wedge\ \ (isValid\ n_{t_{now}})
\end{array}}{\begin{array}{l}
\forall\, t_1 \ \mid \mathsf{latest}(n.eFirst, t_{now} - a) \leq t_1 \leq \mathsf{earliest}(n.eLast, t_{now} - i) \\
\qquad \iff (\ D_{[t_1 \ldots t_{now}]} \in [[^{i,a} p_k]]^L \\
\qquad\qquad \wedge\ \ isValid\ (n.predec)_{t_1}\ )
\end{array}} \quad (6.7) \\
\hline
\begin{array}{l}
\exists\, t_2 \bullet D_{[t_2 \ldots t_{now}]} \in [[^{i,a} p_k]]^L \\
\qquad \wedge\ \ isValid\ (n.predec)_{t_2}
\end{array} \\
\hline
\begin{array}{l}
\exists\, t_2 \bullet D_{[t_2 \ldots t_{now}]} \in [[^{i,a} p_k]]^L \\
\qquad \wedge\ \ D_{[t_{startSess} \ldots t_2]} \in [[\mathsf{SPath}\ n.predec]]^L
\end{array} \\
\hline
D_{[t_{startSession} \ldots t_{now}]} \in [[(\mathsf{SPath}\ \ n.predec)\ \underline{;}\ n.expHd]]^L \\
\hline
D_{[t_{startSession} \ldots t_{now}]} \in [[\mathsf{SPath}\ n]]^L
\end{array}
\tag{6.34}
$$

6.3.4 Prime Nodes Representing Partial Interpretations w.r.t. Sub-Expressions

In this section only one-level conjuction are considered, i.e. conjunctions of chop sequences containing only $^{i,a}p_k$ expressions. In a second step `AND/OR` expressions will be embedded in these sequences recursively, cf. section 6.3.5.

Principally, all nodes n are connected to a predecessor node by $n.predec$ independently from the hierarchical structure of conjunctions and disjunctions. Additionally each node which represents a specification which is sub-expression of an `AND/OR` expression refers by its attribute $.livesIn$ to one certain `OrGr`-object, which in turn refers by its attribute $.tstPartOf$ to one certain `ATst`-node, cf. the informal description in section 4.7.8.

Let n_k be a non-top level node which is valid at t_{now}. Let n_0 be its leading node as defined in 4.7.8, i.e. the distinct node in $(predec^*)\,(n_k)$, which is the leftmost node referring to the same `OrGr`-object as n_k.

Then $\langle n_0, \ldots, n_k \rangle$ is the node chain connecting n_0 and n_k by $predec^{\sim}$, and $\mathsf{SPath}\ n_0 \rightsquigarrow n_k \in \mathcal{L}S'$ is the specification term resulting from the chop-wise concatenation of the specification particles corresponding to these nodes, i.e. $n_0\,.expHd\ \underline{;} \ldots \underline{;}\ n_k\,.expHd$.

Further I_x and A_x shall stand for the minimal and maximal duration requirements of $n_x.expHd = {}^{I_x,A_x}p_{\_}$.

The being valid of a certain `Prime` node implies the existence of interpretations of the *complete* trace data D w.r.t. a specification e_C constructed by the chop-wise concatenation of the specification particles corresponding to all predecessor nodes starting with a successor of n_{-1}, as demonstrated above, and formulated in (6.27).

This, naturally, implies the existence of some interpretations of some suffices of D w.r.t. all suffices of the chop sequence e_C. In the general case, the information required for constructing these interpretations is *not* represented by node objects.

But in case of non-top level *LNode* objects the algorithm maintains four additional time-valued data fields, which allow to derive immediately informations from any such node n_k about the fulfillment of a suffix of the test data D w.r.t. the sub-specification SPath $n_0 \rightsquigarrow n_k$.

This can be demonstrated by an induction going backward in time from n_k to n_0, where for each node n_x of this chain the fundamental local interpretation of each node (6.7) is instantiated for the predecessor node n_{x-1}, by substituting for the single value t_{now} the whole *interval*, in which n_{x-1} is known as having been valid.

For easier reading of the following derivation, note that each time valued variable t_x stands for the possible start time of a segment represented by n_x, and t_{x+1} for the possible end time. Thus t_{x+1} corresponds to t_{now} from (6.7), when this property is "instantiated in the past".

$$
\begin{array}{l}
\forall\, t_{now}, n_k : Prime \mid isValid\ n_{k,t_{now}} \bullet \\
\quad \forall\, t_k \mid \mathsf{latest}(n_k.eFirst, t_{now} - A_k) \le t_k \le \mathsf{earliest}(n_{k,t_{now}}.eLast, t_{now} - I_k) \\
\qquad \Longleftrightarrow D_{[t_k \ldots t_{now}]} \in [[^{I_k,A_k} p_k]]^L \\
\qquad\qquad \wedge\ isValid\ (n_k.predec) \\
\qquad\qquad \overline{\qquad\qquad\qquad\qquad} (6.7) \\
\qquad\qquad \forall\, t_{k-1} \mid \mathsf{latest}(n_{k-1}.eFirst, t_k - A_{k-1}) \\
\qquad\qquad\qquad \le t_{k-1} \le \mathsf{earliest}(n_{k-1,\boxed{t_k}}.eLast, t_k - I_{k-1}) \\
\qquad\qquad\qquad \Longleftrightarrow D_{[t_{k-1} \ldots t_k]} \in [[^{I_{k-1},A_{k-1}} p_{k-1}]]^L \\
\qquad\qquad\qquad\qquad \wedge\ isValid\ (n_{k-1}.predec) \\
\qquad\qquad \overline{\qquad\qquad\qquad\qquad} (6.39) \\
\qquad\qquad \forall\, t_{k-1} \mid \mathsf{latest}(n_{k-1}.eFirst, t_k - A_{k-1}) \\
\qquad\qquad\qquad \le t_{k-1} \le \mathsf{earliest}(n_{k-1,t_{now}}.eLast, t_k - I_{k-1}) \\
\qquad\qquad\qquad \Longleftrightarrow D_{[t_{k-1} \ldots t_k]} \in [[^{I_{k-1},A_{k-1}} p_{k-1}]]^L \\
\qquad\qquad\qquad\qquad \wedge\ isValid\ (n_{k-1}.predec) \\
\qquad \overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \textit{(substitute lower and upper bounds of } t_k \textit{ and collect consequences.)} \\
\qquad \forall\, t_{k-1} \mid \mathsf{latest}(n_{k-1}.eFirst, \mathsf{latest}(n_k.eFirst, t_{now} - A_k) - A_{k-1}) \\
\qquad\qquad \le t_{k-1} \\
\qquad\qquad \le \mathsf{earliest}(n_{k-1,t_{now}}.eLast, \mathsf{earliest}(n_{k,t_{now}}.eLast, t_{now} - I_k) - I_{k-1}) \\
\qquad\qquad \Longleftrightarrow D_{[t_{k-1} \ldots t_k]} \in [[^{I_{k-1},A_{k-1}} p_{k-1}]]^L \ \wedge\ D_{[t_k \ldots t_{now}]} \in [[^{I_k,A_k} p_k]]^L \\
\qquad\qquad\qquad \wedge\ isValid\ (n_{k-1}.predec) \\
\qquad \overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \textit{(apply distributivity and associativity of } \mathsf{min/max}\textit{, and join the consequences.)} \\
\qquad \forall\, t_{k-1} \mid \mathsf{latest}(n_{k-1}.eFirst, n_k.eFirst - A_{k-1}, t_{now} - (A_{k-1} + A_k)) \\
\qquad\qquad \le t_{k-1} \\
\qquad\qquad \le \mathsf{earliest}(n_{k-1,t_{now}}.eLast, n_{k,t_{now}}.eLast - I_{k-1}, t_{now} - (I_{k-1} + I_k)) \\
\qquad\qquad \Longleftrightarrow D_{[t_{k-1} \ldots t_{now}]} \in [[^{I_{k-1},A_{k-1}} p_{k-1} \underline{;}\ ^{I_k,A_k} p_k]]^L \\
\qquad\qquad\qquad \wedge\ isValid\ (n_{k-1}.predec) \\
\hline\hline
\ldots \\
\textit{(repeat this derivation until } n_0 \textit{ is reached.)} \\
\ldots \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\forall\, t_0 \mid \mathsf{latest}(\ n_0.eFirst, \\
\qquad\qquad n_1.eFirst - A_0, \\
\qquad\qquad n_2.eFirst - (A_0 + A_1), \\
\qquad\qquad \ldots, \\
\qquad\qquad n_k.eFirst - (A_0 + \ldots + A_{k-1}), \\
\qquad\qquad t_{now} - (A_0 + \ldots + A_k)) \\
\qquad \le t_0 \\
\qquad \le \mathsf{earliest}(\ n_{0,t_{now}}.eLast, \\
\qquad\qquad n_{1,t_{now}}.eLast - I_0, \\
\qquad\qquad n_{2,t_{now}}.eLast - (I_0 + I_1), \\
\qquad\qquad \ldots, \\
\qquad\qquad n_{k,t_{now}}.eLast - (I_0 + \ldots + I_{k-1}), \\
\qquad\qquad t_{now} - (I_0 + \ldots + I_k)) \\
\quad \Longleftrightarrow D_{[t_0 \ldots t_{now}]} \in [[\mathsf{SPath}\ n_0 \rightsquigarrow n_k]]^L \\
\qquad \wedge\ isValid\ (n_0.predec)
\end{array}
\tag{6.35}
$$

The non-emptiness of all intervals given by the expressions "$\mathsf{latest}(\ldots) \leq t \leq \mathsf{earliest}(\ldots)$" is implied by the fact that all those expressions only comprehend (possibly infinitely many!) legal instantiations of (6.7), for which the non-emptiness has been shown in the general case.

A central pre-requisite for an efficient implementation is the fact that we do not need to know the "former" values of $n_{x-1,t_x}.eLast$, in spite of their appearance when simply instantiating (6.7) at all former time instants $t_k, t_{k-1}, \ldots$, as indicated with the red box $n_{k-1,\boxed{t_k}}.eLast, \ldots$ in the first instantiation step above.

Instead, it is true for every node n_x from $\langle n_0, \ldots, n_{k-1} \rangle$ that we can always use the current value $n_{x,t_{now}}.elast$! This follows from the definition of $eLast$ and the characteristics of its behaviour.

Three different cases have to be distinguished:

In the first case the predecessor of node n_x had already left its valid state at the former time t_x. Since this can happen only once in its life-time, it follows that ...

$$\begin{array}{l} \forall\, n \in Prime \mid isValid\ n_{t_{now}} \bullet \\ \quad t \leq t_{now} \ \wedge\ n_t.eLast \neq \infty \implies n_t.eLast = n_{t_{now}}.eLast \end{array} \tag{6.36}$$

In the second case $n_x.predec$ is has been valid at t_x and still is at t_{now}. Then it holds that

$$n_{x,t_x}.eLast = \infty = n_{x,t_{now}}.eLast \tag{6.37}$$

The third case is given by $isValid_{t_x}(n_x.predec) \wedge \neg\ isValid_{t_{now}}(n_x.predec)$. So $n.predec$ has left its valid state at some time instant $t_x \leq t_p \leq t_{now}$, and $n.eLast$ has been set to t_p. So it follows that ...

$$\begin{array}{l} \forall\, n_x : Prime, t_{now}, t_x : \mathbb{T} \mid isValid_{t_x}(n_x.predec) \wedge \neg\ isValid_{t_{now}}(n_x.predec) \bullet \\ \quad n_{t_{now}}.eLast \geq t_x \end{array} \tag{6.38}$$

In this third case, in spite of the values of $.eLast$ being different at the two different time instants, all expressions of kind "$\mathsf{earliest}(\ldots)$" in formula (6.35) yield identical values in both cases :

$$\begin{array}{l} \forall\, n \in Prime, t_0, t_{now} : \mathbb{T}, I_x : \mathbb{T}' \mid t_0 \leq t_{now} \wedge I_x \geq 0.0 \\ \qquad\qquad \wedge\ isValid_{t_0}(n.predec) \wedge \neg\ isValid_{t_{now}}(n.predec) \bullet \\ \quad \begin{array}{rl} n_{t_0}.eLast = \infty & \\ \hline & t_0 < n_{t_0}.eLast \\ \wedge & t_0 \leq n_{t_{now}}.eLast \qquad (6.38) \\ \hline & t_0 - I_x < n_{t_0}.eLast \\ \wedge & t_0 - I_x \leq n_{t_{now}}.eLast \\ \hline & \mathsf{earliest}(n_{t_0}.eLast, t_0 - I_x) = t_0 - I_x \\ \wedge & \mathsf{earliest}(n_{t_{now}}.eLast, t_0 - I_x) = t_0 - I_x \\ \hline & \mathsf{earliest}(n_{t_0}.eLast, t_0 - I_x) = \mathsf{earliest}(n_{t_{now}}.eLast, t_0 - I_x) \end{array} \end{array} \tag{6.39}$$

Reading this result with the substitutions $n_{k-X}/n_0, I_{k-X}/I_0, t_{k-X+1}/t_0$ shows the correctness of the simplification applied in each step of (6.35).

For the sake of efficient calculation, the algorithm maintains four attributes of *LNode* objects which serve as cache variables. The first two of these do depend only on the specification terms and thus can be assigned a value once, when creating the new node object, cf. $newNode()^{(5.35)}$ and $calcMin/MaxPreds()^{(5.36)}$. These attributes and their data type are ...

- $n.sumMinPreds : \mathbb{D}$ sums up the minimal duration requirements of all predecessor nodes down to and including the leading node of n.
- $n.sumMaxPreds : \mathbb{D}_+$ does the same for the maximal duration requirements.

Using the same index notation as in (6.35), this can informally be written as ...

$$\begin{array}{lcl} n_k.sumMinPreds & = & I_0 + I_1 + \ldots + I_{k-1} \\ n_k.sumMaxPreds & = & A_0 + A_1 + \ldots + A_{k-1} \end{array} \tag{6.40}$$

Substituting these values into the result of (6.35), we get ...

$$\begin{array}{l} \forall\, t_0 \mid \mathsf{latest}(\; n_0.eFirst, \\ \qquad\qquad n_1.eFirst - n_1.sumMaxPreds, \\ \qquad\qquad n_2.eFirst - n_2.sumMaxPreds, \\ \qquad\qquad \ldots, \\ \qquad\qquad n_k.eFirst - n_k.sumMaxPreds, \\ \qquad\qquad t_{now} - A_k - n_k.sumMaxPreds) \\ \qquad \leq t_0 \\ \qquad \leq \mathsf{earliest}(\; n_{0,t_{now}}.eLast, \\ \qquad\qquad\qquad n_{1,t_{now}}.eLast - n_1.sumMinPreds, \\ \qquad\qquad\qquad n_{2,t_{now}}.eLast - n_2.sumMinPreds, \\ \qquad\qquad\qquad \ldots, \\ \qquad\qquad\qquad n_{k,t_{now}}.eLast - n_k.sumMinPreds, \\ \qquad\qquad\qquad t_{now} - I_k - n_k.sumMinPreds) \\ \qquad \Longleftrightarrow\; D_{[t_0 \ldots t_{now}]} \in [[\mathsf{SPath}\; n_0 \rightsquigarrow n_k]]^L \\ \qquad\qquad \wedge\; isValid\;(n_0.predec) \end{array} \tag{6.41}$$

Since for each node n_x with $0 \leq x < k$ these calculations have only to be done once, but n_k may have arbitrarily many successor nodes, the expressions above are again folded into the calculation of the values of two additional cache variables *seFirst* and *seLast*, — read: "Sequence Entry First" and "Sequence Entry Last", — the definition of which can informally be described as ...

$$\begin{array}{lcl} n_0.seFirst & = & n_0.eFirst \\ n_x.seFirst & = & \mathsf{latest}(n_{x-1}.seFirst, n_x.eFirst - n_x.sumMaxPreds) \\ n_{0,t}.seLast & = & n_{0,t}.eLast \\ n_{x,t}.seLast & = & \mathsf{earliest}(n_{x-1,t}.seLast, n_{x,t}.eLast - n_x.sumMinPreds) \end{array} \tag{6.42}$$

Therefore the result of (6.35) is further simplified to ...

$$\begin{aligned} \forall\, t_0 \mid\ & \mathsf{latest}(\ n_k.seFirst, t_{now} - A_k - n_k.sumMaxPreds) \\ & \leq t_0 \\ & \leq \mathsf{earliest}(\ n_{k,t_{now}}.seLast, t_{now} - I_k - n_k.sumMinPreds) \\ & \Longleftrightarrow\ (\ D_{[t_0 \ldots t_{now}]}\ \in\ [[\mathsf{SPath}\ n_0 \rightsquigarrow n_k]]^L \\ & \qquad \wedge\ \ isValid\ \ (n_0\ .predec)\ ) \end{aligned} \tag{6.43}$$

The value of $.seFirst$ behaves like $.eFirst$, i.e. it does never change throughout the life-time of a valid $Node$ object. Therefore, the value of $.seFirst$ can be calculated once, in the same evaluation step as $.eFirst$, i.e. when the external predicate goes `true`. This is done in the function $RNode\_signalRaises()^{(5.20)}$.

Contrarily, the value of $n_{k,t}.eLast$ of a valid $Prime$-node will be changed from ∞ to the current time instant, as soon as $n_k.predec$ terminates. Therefore the value of $n_{k,t}.seLast$ can change arbitrarily often, potentially in all those time instants when a value of $n_{x,t}.eLast$ changes, with $x \leq k$.

This is realized in the algorithm as follows:

Whenever a node n terminates, this event is signaled to all its successors n_S by applying to them the function $RNode\_becomesFixed()^{(5.49)}$.This function not only assigns the current time to $n_S.eLast$, but also calculates a new candidate value for $n_S.seLast$.

This value is passed to the function $LNode\_SEL\_lowers()^{(5.51)}$ which is called on n_S. This function checks if this candidate value is indeed a harder constraint, i.e. an earlier value than the current $n_S.seLast$. If so, this attribute is overwritten, and the value is signaled to all those of the successors of n_S which belong to the same node chain[4] by applying the function $LNodeSet\_SEL\_lowers()^{(5.51)}$ recursively.

6.3.5 Construction of ASol Nodes Representing the Solutions of Conjunctions

.

The properties from (6.43) and (6.81) are now used to construct the semantics of an `ASol` node, which represents the complete set of segments which all fulfill a certain conjunction of sub-specifications.

As soon as some node n becomes valid which is a final node (i.e. a node which represents a partial interpretation w.r.t. some linearization of a *complete* sub-expressions of an `AND/OR` expression, as defined in section 4.7.8 above[5]) and iff all other `OrGr`-objects (which belong to the same `ATst`-node as the `OrGr` containing n) contain also at least one final node, then for all possible combinations of one final

[4] ...and possibly to all those `ASol`-nodes which use n as a part of their represented solution, see below in section 6.3.5.

node from each `OrGr`, one new `ASol`-node N each is constructed, representing the fulfillment of the conjunction of the sub-specifications represented by these nodes.

The schema definition corresponding to the `ASol`-nodes is given in formula (5.6), and the function $installSolution()^{(5.42)}$ creates that new `ASol` object N. All attribute values of N can be calculated in this function, and all of these, except $.aeLast$ and $.seLast$, will stay constant.[6]

$N.solParts$ is set to the collection of references to those final nodes the combination of which caused the creation of N.

The first and last starting point of the segments represented by an `ASol` node N are set to the latest and earliest values of all final nodes contained in $N.solParts$, and the values $N.minSubSum$ and $N.maxSubSum$ cache the maximal value of all minimal duration requirements and the minimal value of all maximal duration requirements of all sub-expressions contained in $N.solParts$:

$$\begin{array}{lll} \forall N : \texttt{ASol}, t : \mathbb{T} \mid & N.solParts = \{m_0, \ldots, m_k\} \bullet & \\ N.aeFirst & = & \mathsf{latest}\,\{m_0.seFirst, \ldots, m_k.seFirst\} \\ N.aeLast_t & = & \mathsf{earliest}\,\{m_0.seLast_t, \ldots, m_k.seLast_t\} \\ N.minSubSum & = & \mathsf{max}\,(\!|(\, m_0.sumMinPreds + minDura(m_0), \\ & & \qquad \ldots, m_k.sumMinPreds + minDura(m_k)\,) \\ N.maxSubSum & = & \mathsf{min}\,(\!|(\, m_0.sumMaxPreds + maxDura(m_0), \\ & & \qquad \ldots, m_k.sumMaxPreds + maxDura(m_k)\,) \end{array} \tag{6.44}$$

The value of $N.aeLast$ is dynamic, because it depends on the dynamic values of $.seLast$ of all nodes from $N.solParts$, and the value of $.seLast$ is dynamic, because each `ASol` node is treated as any $LNode$ w.r.t. the calculation of $.seLast$ as described above in section 6.3.4.

When trying to create an `ASol` node N at time instant t_{now}, the function $installSolution()$ additionally tests the conditions

$$\begin{array}{lll} N.minSubSum & \leq & N.maxSubSum \\ N.aeFirst & \leq & N.aeLast_{t_{now}} \end{array} \tag{6.45}$$

Therefore combinations of sub-expressions which are *per se* fulfilled, but which do conflict in their timing or duration requirements will not yield an `ASol`-node.

[5] This fact is detected in the algorithm in chapter 5 using the special terminal symbol $\triangle_\triangle$ appended to the end of each such sub-expression, cf. function $termInst()^{(5.33)}$, which calls $LNode\_endReached()^{(5.40)}$ in the case that $\triangle_\triangle$ is element of the set of subsequent expressions.

[6] Basically $N.seLast$ and $N.seFirst$ behave like the same attributes in a `Prime` node. This section treats only one-level `AND/OR` expressions, and therefore `ASol` nodes are not yet part of node chains which represent sub-expressions. The behaviour of $N.seFirst$ and $N.seLast$ is related to such nested expressions, and is treated below in the dedicated section 6.3.7.

Corresponding to the definition in (6.6) on page 59 for `Prime` nodes, for `ASol` nodes the following derived dynamic predicate is defined:

$$\begin{array}{l} \forall N : \texttt{ASol}, t : \mathbb{T} \bullet \\ \quad isValid\ N_t \stackrel{def}{=} (\forall n \in N.solParts \bullet isValid\ n_t) \\ \qquad\qquad \wedge\ N.aeFirst + N.minSubSum \leq t \\ \qquad\qquad \wedge\ t \leq N_t.aeLast + N.maxSubSum \end{array} \tag{6.46}$$

The validity of this predicate is realized in the implementation by (a) installing and modifying up to two timer requests for N, and (b) by two mechanisms which signal to N all relevant changes in the state of any node in $N.solParts$.

(a)

Let t_{now} be the time instant when N is created.

If $N.aefirst + N.minSubSum \leq t_{now}$ holds, then the node is put directly into a valid state.

Otherwise it is put into the time-in state, and a time-in request for the future time instant $N.aeFirst + N.minSubSum$ in initiated.

If both $N.aeLast$ and $N.maxSubSum$ are $\neq \infty$, a time-out request is initiated for the time instant $N.aeLast + N.maxSubSum$.

(b)

Whenever an *LNode* n terminates, the function $LNode\_terminates()^{(5.47)}$ calls itself recursively for all `ASol` nodes which use n as a part of their solution.

As described in the preceding section, the function $LNode\_SEL\_lowers()^{(5.51)}$ is recursively called for possibly calculating a new and earlier value for $n.seLast$ whenever some transitive predecessor in the same node chain terminates.

If this function reaches a final node n, the function $ASol\_subSEL\_lowers()^{(5.52)}$ is called for all `ASol` nodes which use n as a part of their solution. This possibly updates the value $.aeLast$ in these `ASol` nodes to a new and earlier value. The currently pending time-out request has to be adjusted accordingly, which is discussed in detail in section 6.3.5.1.

If $LNode\_terminates()$ is called for N before the time-in request has expired, N is simply discarded. Otherwise, the expiration of the time-in request lets N transit to a valid state. This guarantees that $isValid\ (N_t) \Longrightarrow t \geq N.aeFirst + N.minSubSum$

The second transition of N is from the valid to the terminated state. This is performed either when the time-out request expires, guaranteeing that $isValid\ (N_t) \Longrightarrow t \leq N_t.aeLast + N.maxSubSum$ always holds, or when any of the nodes from $N.solParts$ terminates, signaled to N as described above and guaranteeing that $isValid\ (N_t) \Longrightarrow \forall n \in N.solParts \bullet isValid(n_t)$.

Any `ASol`-node which enters the valid state behaves like any *Prime*-node: The set of subsequent expressions is calculated, and for each of its members one new *LNode* is created in the testing state. This is realized by $LNode\_becomesValid()^{(5.31)}$ being called for N in the same way as for any `Prime` node. This call is either performed directly by $installSolution()$, or by the scheduling function $execute\_min^{(5.19)}$ in case of a time-in request.

The fundamental local property of each `ASol`-node, analog to (6.7) on page 59 for `Prime`-nodes, is given by the lemma ...

$$
\begin{array}{lll}
\forall\, t_{now}, t : \mathbb{T}\,, N : \texttt{ASol} \mid & isValid\ N_{t_{now}} & \\
 & \wedge\ t_0 = \mathsf{latest}(n.aeFirst, t_{now} - N.maxSubSum) & \\
 & \wedge\ t_2 = \mathsf{earliest}(\,(N_{t_{now}}).aeLast, t_{now} - N.minSubSum) & \\
\bullet \quad t_0 \leq t \leq t_2 & \Longleftrightarrow \quad \big(\, D_{[t \ldots t_{now}]} \in [[\mathsf{expr}\, N]]^L \ \wedge\ isValid\ (n.predec)_t \big) & \\
\wedge \quad t_0 \leq t_2 & &
\end{array}
\tag{6.47}
$$

In this section this property can be demonstrated to hold for all `ASol` nodes representing `AND/OR` expressions containing only *Prime* nodes, based on the results of the preceding paragraph, which considered sub-expressions with the same restriction.

Let for a certain N : `ASol` be $N.solParts = \{n_0, \ldots, n_k\}$, and for each $0 \leq x \leq k$ the node m_x be the leading node of n_x.

Then the expression from $\mathcal{L}\,S''$ for which a segment of a partial interpretation is represented by N is given by ...

$$\mathsf{expr}N \ = \ \texttt{AND}\,(\mathsf{SPath}\ m_0 \rightsquigarrow n_0, \ldots, \mathsf{SPath}\ m_k \rightsquigarrow n_k) \tag{6.48}$$

...and the corresponding semantics as ...

$$[[\,\mathsf{expr}N\,]]^L \ = \ \bigcap \langle\, [[\mathsf{SPath}\ m_0 \rightsquigarrow n_0]]^L, \ldots, [[\mathsf{SPath}\ m_k \rightsquigarrow n_k]]^L \,\rangle \tag{6.49}$$

Since each leading m_x serves as the reference for calculating $n_x.seFirst/.seLast$, and since for each $n \in N.solparts$ it holds that $n.predec = N.predec^{(5.42)}$, the ($\Longrightarrow$) statement contained in (6.47) follows from (6.43) by these derivations:

$$
\begin{array}{l}
\forall\, t_{now} : \mathbb{T}\,, N : \texttt{ASol}, n : LNode \;\mid\; n \in N.solParts \;\wedge\; n.expr =^{I,A} p_{\_} \;\bullet \\
isValid\ N_{t_{now}} \\
\hline
isValid\ n_{t_{now}} \\
\hline
\forall\, t_0 \mid \mathsf{latest}(\ n_k.seFirst, t_{now} - (n_k.sumMaxPreds + A)) \\
\qquad \leq t_0 \\
\qquad \leq \mathsf{earliest}(\ n_{k,t_{now}}.seLast, t_{now} - (n_k.sumMinPreds + I)) \\
\qquad \Longleftrightarrow\ (\ D_{[t_0 \ldots t_{now}]} \in [[\mathsf{SPath}\ n_0 \rightsquigarrow n_k]]^L \\
\qquad\qquad \wedge\ isValid\ (n_0.predec)\,) \\
\hline
\forall\, t_0 \mid \mathsf{latest}(\ n_k.seFirst, t_{now} - N.maxSubSum) \\
\qquad \leq t_0 \\
\qquad \leq \mathsf{earliest}(\ n_{k,t_{now}}.seLast, t_{now} - N.minSubSum) \\
\qquad \Longrightarrow\ (\ D_{[t_0 \ldots t_{now}]} \in [[\mathsf{SPath}\ n_0 \rightsquigarrow n_k]]^L \\
\qquad\qquad \wedge\ isValid\ (n_0.predec)\,)
\end{array}
\quad
\begin{array}{l}
n.sumMaxPreds + A \;\geq\; N.maxSubSum \\
n.sumMinPreds + I \;\leq\; N.minSubSum
\end{array}
$$

$$
\begin{array}{l}
\hline
\forall\, t_0 \mid \mathsf{latest}(\ N.aeFirst, t_{now} - N.maxSubSum) \\
\qquad \leq t_0 \\
\qquad \leq \mathsf{earliest}(\ N_{t_{now}}.aeLast, t_{now} - N.minSubSum) \\
\qquad \Longrightarrow\ (\ D_{[t_0 \ldots t_{now}]} \in [[\mathsf{SPath}\ n_0 \rightsquigarrow n_k]]^L \\
\qquad\qquad \wedge\ isValid\ (n_0.predec)\,)
\end{array}
\quad
\begin{array}{l}
n_k.seFirst \leq N.aeFirst \\
n_{k,t_{now}}.seLast \geq N_{t_{now}}.aeLast
\end{array}
\tag{6.50}
$$

The ($\Longleftarrow$) statement contained in (6.47) can be shown be rewriting the ($\Longleftarrow$)-part of (6.43) as follows:

$$
\begin{array}{l}
\forall\, t_0 \mid\ t_0 < \mathsf{latest}(\ n_k.seFirst, t_{now} - (n_k.sumMaxPreds + A_k)) \\
\qquad \Longrightarrow\ D_{[t_0 \ldots t_{now}]} \notin [[\mathsf{SPath}\ n_0 \rightsquigarrow n_k]]^L \\
\qquad\qquad \vee\ \neg\ isValid\ (n_0.predec) \\
\forall\, t_0 \mid\ \mathsf{earliest}(\ n_{k,t_{now}}.seLast, t_{now} - (n_k.sumMinPreds + I_k)) \;<\; t_0 \\
\qquad \Longrightarrow\ D_{[t_0 \ldots t_{now}]} \notin [[\mathsf{SPath}\ n_0 \rightsquigarrow n_k]]^L \\
\qquad\qquad \vee\ \neg\ isValid\ (n_0.predec)
\end{array}
\tag{6.51}
$$

This proposition on the upper and lower limits of t_0 can be demonstrated by considering two different cases each.

For the **lower** limits these cases are ...

$$\forall\, t_{now} : \mathbb{T}\,,\, N : \texttt{ASol} \bullet$$

$$t_{now} - N.maxSubSum \leq N.aeFirst$$

$$\exists\, n_k \in N.solParts \mid n_k.seFirst = N.aeFirst \;\wedge\; maxDura(n_k) = A$$

(6.51)

$$\forall\, t_0 \mid t_0 < \mathsf{latest}(\, n_k.seFirst, t_{now} - (n_k.sumMaxPreds + A_k))$$
$$\Longrightarrow D_{[t_0 \ldots t_{now}]} \notin [[\mathsf{SPath}\ n_0 \rightsquigarrow n_k]]^L$$
$$\vee\ \neg\ isValid\ (n_0.predec)$$

$t_{now} - (n_k.sumMaxPreds + A_k) \leq$
$t_{now} - N.maxSubSum \leq$
$N.aeFirst = n_k.seFirst$

$$\forall\, t_0 \mid t_0 < n_k.seFirst$$
$$\Longrightarrow D_{[t_0 \ldots t_{now}]} \notin [[\mathsf{SPath}\ n_0 \rightsquigarrow n_k]]^L$$
$$\vee\ \neg\ isValid\ (n_0.predec)$$

$$\forall\, t_0 \mid t_0 < N.aeFirst$$
$$\Longrightarrow D_{[t_0 \ldots t_{now}]} \notin [[\mathsf{expr}\, N]]^L$$
$$\vee\ \neg\ isValid\ (n_0.predec)$$

(6.52)

...and in the other case ...

$$\forall\, t_{now} : \mathbb{T}\,,\, N : \texttt{ASol} \bullet$$

$$N.aeFirst \leq t_{now} - N.maxSubSum$$

$$\exists\, n_k \in N.solParts \mid maxDura(n_k) = A \;\wedge\; n_k.sumMaxPreds + A = N.maxSubSum$$

(6.51)

$$\forall\, t_0 \mid t_0 < \mathsf{latest}(\, n_k.seFirst, t_{now} - (n_k.sumMaxPreds + A_k))$$
$$\Longrightarrow D_{[t_0 \ldots t_{now}]} \notin [[\mathsf{SPath}\ n_0 \rightsquigarrow n_k]]^L$$
$$\vee\ \neg\ isValid\ (n_0.predec)$$

$n_k.seFirst \leq N.aeFirst \leq$
$t_{now} - N.maxSubSum =$
$t_{now} - (n_k.sumMaxPreds + A_k)$

$$\forall\, t_0 \mid t_0 < t_{now} - (n_k.sumMaxPreds + A_k)$$
$$\Longrightarrow D_{[t_0 \ldots t_{now}]} \notin [[\mathsf{SPath}\ n_0 \rightsquigarrow n_k]]^L$$
$$\vee\ \neg\ isValid\ (n_0.predec)$$

$$\forall\, t_0 \mid t_0 < t_{now} - N.maxSubSum$$
$$\Longrightarrow D_{[t_0 \ldots t_{now}]} \notin [[\mathsf{expr}\, N]]^L$$
$$\vee\ \neg\ isValid\ (n_0.predec)$$

(6.53)

These two results can be combined to ...

$$\forall\, t_0 \mid t_0 < \mathsf{latest}(t_{now} - N.maxSubSum, N.aeFirst)$$
$$\Longrightarrow D_{[t_0 \ldots t_{now}]} \notin [[\mathsf{expr}\, N]]^L \qquad (6.54)$$
$$\vee\ \neg\ isValid\ (n_0.predec)$$

For the upper limits given by $N_{t_{now}}.aeLast$ and $t_{now} - N.minSubSum$ the same proof pattern can be applied accordingly.

6.3.5.1 Proof of the Non-Emptiness of $\{t_0 \ldots t_2\}$

To show the **non-emptiness** of the set of time instants at which the segments represented by an ASol-node do start, we have to show that

$$\begin{aligned} &\forall\, t_{now} : \mathbb{T}, N : \texttt{ASol} \bullet \\ &\quad isValid\ N_{t_{now}} \implies \mathsf{latest}(N.aeFirst, t_{now} - N.maxSubSum) \\ &\qquad\qquad\qquad\qquad \leq \mathsf{earliest}(N_{t_{now}}.aeLast, t_{now} - N.minSubSum) \end{aligned} \tag{6.55}$$

This can be demonstrated by considering all possible four combinations:

$$t_{now} - N.maxSubSum \leq t_{now} - N.minSubSum \tag{6.56}$$

...is checked once when constructing the ASol-node.

$$N.aeFirst \leq t_{now} - N.minSubSum \tag{6.57}$$

...is true because the node does not enter the *valid* state earlier than the time instant $N.aeFirst + N.minSubSum$, when the corresponding min-timer expires.

More complex are the inequalities involving $N_{t_{now}}.eaLast$, because this value behaves dynamically: It always is set to the earliest() value of the $.seLast$-values of all nodes contained in $N.solParts$, cf. (6.44) on page 72,

At the time of construction t_c, it is checked once by the function $installSolution()^{(5.42)}$ that ...

$$N.aeFirst \leq N_{t_c}.aeLast \tag{6.58}$$

Additionally, if $N.maxSubSum < \infty$, a max-timer request is installed, which expires at $N_{t_c}.aeLast + N.maxSubSum$, the expiration of which will terminate the valid state. Therefore initially the validity of N implies ...

$$t_{now} - N.maxSubSum \leq N_{t_c}.aeLast \tag{6.59}$$

But since $N.aeLast$ can lower arbitrarily often due to the further behaviour of the final nodes from $N.solParts$, it has to be shown that (6.58) and (6.59) will hold in all cases.

W.r.t. (6.59) :

Whenever at a time instant t_{now} the value $n.seLast$ of some node contained in $N.solParts$ changes, i.e. lowers, this time-out request imposed on N has to be adjusted accordingly, for guaranteeing (6.59) always to hold as long as $isValid(N)$ holds.

This is realized by the function $ASol\_subSEL\_lowers()^{(5.52)}$, which is called whenever a final node in $N.solParts$ lowers its value of $.seLast$, and which sets the new time-out value to at $N_{t_{now}}.aeLast + N.maxSubSum$.

Rewriting (6.59) to $t_{now} \leq N_{t_c}.aeLast + N.maxSubSum$ shows that this property is equivalent to the fact that this timer update is feasible, i.e. that the new value for the time-out expiration $newTO$ does not lie "in the past of the execution", i.e. that $newTO \geq t_{now}$ holds.

This can be shown as follows:

Whenever a new value $newAEL$ for $N.eaLast$ is calculated[7] at a time instant t_{now}, this is always caused by a node m which is directly contained in one of the sub-chains leading to a node from $N.solParts$, and which sets its own $m.seLast$ to this new value $newAEL = t_{now} - \delta$. Then $newTO$ is calculated as $newTO = newAEL + N.maxSubSum$.

Since the recursive case, in which an `ASol` node is contained in a sub-sequence of an `ASol` node, is not yet considered[8], m is a `Prime`-node the predecessor of which terminates. Then δ is equal to $m.sumMinPreds + minDura(m)$, according to the function $LNode\_SEL\_lowers()^{(5.51)}$.

Because of the initially tested invariants of the `ASol` node in (6.45), the definition of $N.minSubSum$ (6.44) and because of (6.56) it holds that

$$
\begin{array}{lcll}
N.minSubSum & \geq & (m.sumMinPreds + minDura(m)) & \textit{(6.44)} \\
newAEL & = & t_{now} - (m.sumMinPreds + minDura(m)) & \textit{(5.47)} \\
\hline
newAEL & \geq & t_{now} - N.minSubSum & \\
N.minSubSum & \leq & N.maxSubSum & \textit{(6.45)} \\
\hline
newAEL & \geq & t_{now} - N.maxSubSum & \\
newTO & = & newAEL + N.maxSubSum & \textit{(5.52)} \\
\hline
newTO & \geq & t_{now} - N.maxSubSum + N.maxSubSum & \\
\hline
newTO & \geq & t_{now} &
\end{array}
\tag{6.60}
$$

Therefore all new, lower time-out requests created at time instant t_{now} will never refer to a time instant which has already passed, and, conversely, each time-out request will guarantee that the interval $N.aeFirst \ldots N.aeLast$ will always be non-empty w.r.t. (6.59).

W.r.t. condition (6.58) two cases have to be distinguished:

(1)

As long as N is in the time-in state, this condition can be violated by the upcoming of the new, lower value $newAEL < N.eaFirst$. If so, the interval $N.aeFirst \ldots N.aeLast$ is empty and N does not represent any solution of the conjunction. Consequently, N is discarded totally from the collection of nodes in the first alternative of $ASol\_subSEL\_lowers()^{(5.52)}$.

[7] Of course this $newAEL$ has any influence on $N.aeLast$ only if it is indeed a harder constraint, i.e. earlier value than the current value $N_{t_{now}}.aeLast$. Cf. the description of the propagation of $.seLast$ in section 4.7.9.4 on page 36.

[8] The proofs for this case will follow in section 6.3.7.1 on page 80.

(2)

If the time-in request has expired, and *isValid N* holds, the *newAEL* cannot violate the condition any more:

$$\frac{\dfrac{\dfrac{isValid\ N}{N.aeFirst + N.minSubSum \ \leq\ t_{now}}}{N.aeFirst \ \leq\ t_{now} - N.minSubSum} \qquad \dfrac{(m.sumMinPreds + minDura(m)) \ \leq\ N.minSubSum}{newAEL = t_{now} - (m.sumMinPreds + minDura(m)) \ \geq\ t_{now} - N.minSubSum}}{\dfrac{N.aeFirst \leq\ t_{now} - N.minSubSum \leq newAEL}{N.aeFirst \leq\ N_{t_{now}}.aeLast}} \tag{6.46}$$

(6.61)

6.3.6 Embedding AND/OR Expressions in the Top Level Chop Sequence

Up to now, it has been shown that the every `Prime` nodes represents partial interpretations of the trace's prefix w.r.t. linear specifications which are sequences of atomic predicates ${}^{i,a}p_k$.

The corresponding central semantic property is expressed by formula (6.27) in section 6.3.3 on page 64.

(6.27) has been proved using only the local semantic properties of `Prime` nodes, as given in formula (6.7) in section 6.3.2 on page 59, together with the special treatment of n_{-1}, as described in the induction in section 6.3.3 on page 64.

Further, the semantic properties of `ASol` nodes have been formulated by (6.47) in the preceding section on page 74, and have been proven for all `ASol` nodes which correspond to `AND/OR` expressions from $\mathcal{L}\,S'$, containing atomic predicates only.

Comparing (6.7) and (6.47), both formulæ can be considered to be identical after applying the following renaming[9]:

$n \in$ `Prime`		$N \in$ `ASol`		*abstraction used in the implementation*
$n.eFirst$	$\equiv$	$N.eaFirst$		
$n.eLast$	$\equiv$	$N.eaLast$		
$n.expHd = {}^{I_k,A_k}p_k$	$\equiv$	expr N		
I_k	$\equiv$	$N.minSubSum$	$=$	$minDura()$
A_k	$\equiv$	$N.maxSubSum$	$=$	$maxDura()$

(6.62)

[9] The auxiliary functions *minDura()* and *maxDura()* are defined in (5.22) and (5.37) and realize the abstraction from the class of a node object. They are already used in the implementation, namely for $calcMin/maxPred()^{(5.36)}$ when creating new node objects.

Since (6.27), the central semantic property of node objects representing a top level chop sequence, relies only on (6.7), it does also hold for specifications which are chop sequences containing `AND/OR` expressions, as long as these, in turn, still contain only chop sequences of atomic predicates.

6.3.7 Free Nesting of `AND/OR` Expressions

To allow an arbitrary nesting of `AND/OR` constructions, it has to be shown that the semantic properties of `ASol` nodes as given by (6.47) also hold in case that `ASol` nodes are contained in the node chains.

(6.47) depends mainly on (6.43) on page 71 in section 6.3.4, which describes the semantics of an *LNode* as representing a interpretation suffix.

Additionally, (6.47) depends on (6.56) to (6.59).

(6.43) and (6.56) to (6.59) have been proved for chop sequences made of atomic predicates, i.e. node chains consisting of `Prime` nodes. To allow an arbitrary nesting of `AND/OR` constructions, it has to be demonstrated that they also hold for node chains containing `ASol` nodes.

6.3.7.1 Proof of (6.56) to (6.59) w.r.t. `ASol` nodes

The first two of these propositions depend only on static properties of the `ASol` node N, independent of its contents.

But (6.58) and (6.59) restrict the subsequent lowering of $N.aeLast$, caused by a lowering of $m.seLast$ of some final node from $N.solParts$. Since in this concern any contained `ASol` node behaves differently than a `Prime` node, both properties have to be demonstrated anew.

Let N' be a node contained in a node chain which ends at a final node contained in $N.solParts$. Let N' lower its value of $.aeLast$ at the time instant t_{now} to some value $newAEL'$.

It has been shown in the derivation of (6.60), if N' is an `ASol` node containing only `Prime` nodes, that it holds that ...

$$newAEL' \geq t_{now} - N'.minSubSum \tag{6.63}$$

As defined in the algorithm's function $ASol\_subSEL\_lowers()^{(5.52)}$, a lowering of $N'.aeLast$ can cause a lowering of $N'.seLast$ to a new value $newAEL$, which is propagated up to the final node, and possibly influences the value $N.aeLast$ of the containing `ASol` node:

$$newAEL = newAEL' - N'.sumMinPreds \tag{6.64}$$

From these two properties it follows that

$$newAEL \geq t_{now} - N'.minSubSum - N'.sumMinPreds \tag{6.65}$$

Since $N'.minSubSum$ and $N'.sumMinPreds$ are both summed up into the value of $N.minSubSum$ when creating N (6.44), it holds that ...

$$N'.minSubSum + N'.sumMinPreds \leq N.minSubSum \tag{6.66}$$

...and consequently ...

$$newAEL \geq t_{now} - N.minSubSum \tag{6.67}$$

Using (6.67) like (6.63) above, shows inductively that (6.67) holds for arbitrarily nested AND/OR expressions.

Inserting (6.67) into (6.61) and (6.60) shows that that (6.58) and (6.59) also hold in the arbitrarily nested case.

6.3.7.2 Proof of (6.43) w.r.t. ASol nodes

The derivation of (6.43) as performed in the induction in (6.35) on page 68 relies on the local semantics of Prime nodes, as given by (6.7), and on (6.39).

In the case of node chains mixed from Prime and ASol nodes, (6.47) is equivalent to (6.7) after applying the above-mentioned renaming.

The second pre-requisite, (6.39), is more critical.

It expresses the fact that in all cases occurring in the derivation (6.35) (using the index notation defined therein) for each n_x/N_x in the node chain and for each $t_0 = t_{x+1} \leq t_{now}$ it holds that ...

$$\begin{aligned}
&\mathsf{earliest}(n_{x,t_0}.eLast, t_0 - I_x) &&= \mathsf{earliest}(n_{x,t_{now}}.eLast, t_0 - I_x) \\
&\text{...rewritten for ASol nodes as ...} \\
&\mathsf{earliest}(N_{x,t_0}.aeLast, t_0 - N_x.minSubSum) &&= \mathsf{earliest}(N_{x,t_{now}}.aeLast, \\
&&& \qquad t_0 - N_x.minSubSum)
\end{aligned} \tag{6.68}$$

This allows the above-mentioned and in detail discussed second transformation step in (6.35), which replaces $n_{k-1}.eLast_{t_k}$ by $n_{k-1}.eLast_{t_{now}}$ and is of fundamental importance for the efficiency of the algorithm.

This has shown for Prime nodes in (6.39), but has to be shown for ASol nodes in a different way. E.g. (6.36) does not hold if n is not a Prime node but an ASol node, since $N.aeLast$ can lower its value arbitrarily often, — in contrast to $n.eLast$.

First of all, the previous section has shown (6.67) that whenever a new candidate value $newAEL_t$ for $N.aeLast$ is calculated at some time instant t, it holds that ...

$$newAEL_t \geq t - N.minSubSum \tag{6.69}$$

From this it follows that if a new candidate $newAEL$ for $N.eaLast$ is calculated at some time instant t_2, and t_2 is later than the current value $N_{t_1}.aeLast +$

$N.minSubSum = N_{t_1}.aeLast + N.minSubSum$, this new candidate will never lead to a further lowering of $N.aelast$:

$$\forall\, t_1, t_2 \;\mid\; t_1 < t_2 \;\wedge\; N_{t_1}.aeLast + N.minSubSum \leq t_2 \;\bullet\; \frac{\dfrac{t_2 - N.minSubSum \leq newSEL_{t_2}}{N_{t_1}.aeLast \leq t_2 - N.minSubSum \leq newSEL_{t_2}}}{N_{t_2}.aeLast = \mathsf{earliest}(N_{t_1}.aeLast, newSEL_{t_2}) = N_{t_1}.aeLast} \tag{6.70}$$

As in (6.35), let $n_{x-1} = N$ be some `ASol` node in the node chain $\langle n_0, \ldots, n_k \rangle$. Let $N_{t_x}.aeLast$ be the value of $N.aeLast$ at the time instant t_x, which is referred to by the straight-forward instantiation of (6.7)/(6.47) for some time instant in the past, and let $N_{t_{now}}.aeLast$ be the value used by the algorithm instead of $N_{t_x}.aeLast$.

Additionally, let t be the latest time instant before t_{now} at which the value of $N.eaLast$ has changed.

Then three cases have to be distinguished :

$\forall\, N : \texttt{ASol};\; t_x, t_{now} : \mathbb{T} \mid t_x < t \leq t_{now} \bullet$

(1)

$$\frac{\dfrac{\dfrac{N_{t_x}.aeLast = \infty \;\wedge\; N_{t_{now}}.aeLast \neq \infty}{t_x < N_{t_x}.aeLast \;\wedge\; \dfrac{N_{t_{now}}.aeLast \geq t - N.minSubSum}{N_{t_{now}}.aeLast > t_x - N.minSubSum}}}{\mathsf{earliest}(t_x - N.minSubSum, N_{t_x}.eaLast) = t_x - N.minSubSum \;\wedge\; \mathsf{earliest}(t_x - N.minSubSum, N_{t_{now}}.eaLast) = t_x - N.minSubSum}}{\mathsf{earliest}(t_x - N.minSubSum, N_{t_x}.eaLast) = \mathsf{earliest}(t_x - N.minSubSum, N_{t_{now}}.eaLast)} \tag{6.71}$$

(2)

$$\frac{\dfrac{\dfrac{\dfrac{N_{t_x}.aeLast = T \leq t_x \;\wedge\; t_{now} \leq T + N.minSubSum \;\wedge\; N_{t_{now}}.eaLast \leq T}{t_x < t_{now} \leq T + N.minSubSum \;\wedge\; t_x \leq t_{now} \leq N_{xtn}.eaLast + N.minSubSum}}{t_x - N.minSubSum < T \;\wedge\; t_x - N.minSubSum \leq N_{t_{now}}.eaLast}}{\mathsf{earliest}(t_x - N.minSubSum, N_{t_x}.eaLast) = t_x - N.minSubSum \;\wedge\; \mathsf{earliest}(t_x - N.minSubSum, N_{t_{now}}.eaLast) = t_x - N.minSubSum}}{\mathsf{earliest}(t_x - N.minSubSum, N_{t_x}.eaLast) = \mathsf{earliest}(t_x - N.minSubSum, N_{t_{now}}.eaLast)} \tag{6.72}$$

(3)

$$\frac{\dfrac{N_{t_x}.aeLast \leq t_x \;\wedge\; \dfrac{N_{t_x}.aeLast + N.minSubSum \leq t_{now}}{N_{t_{now}}.aelast = N_{t_x}.aelast}}{\mathsf{earliest}(t_x - N.minSubSum, N_{t_x}.eaLast) = \mathsf{earliest}(t_x - N.minSubSum, N_{t_{now}}.eaLast)}}{} \quad \textit{(6.69)} \tag{6.73}$$

Therefore the critical optimization step in the derivation in (6.35) can be taken also for `ASol` nodes.

6.4 Completeness

In the preceding section the fundamental property of each single node object has been demonstrated: The existence of a valid node object n at some time instant t_{now} indicates that there is an interpretation of the SUT's trace data w.r.t. a certain linear specification (= SPath n) derived from SpecUT.

To show the completeness of the algorithm means to show that, *vice versa*, each partial interpretation causes the existence of at least one valid node object which represents it.

This is not a local property of a single node, but a property of the total collection of nodes, as referred to by *GState.nodes* in chapter 5.

If the SUT's trace data D fulfills SpecUT, then D fulfills some linear specification S_L derived from SpecUT, and there exists (at least) one interpretation i of D w.r.t. S_L. This follows from the respective definitions in section 4.6.

This interpretation splits D into k segments $g_1 \ldots g_k$. Each segment g_m extends from t_m to t_{m+1} and corresponds to an expression e_m, which is either of form ${}^{I_m,A_m}p_m$ or of form AND{}. Additionally it holds that $t_1 = t_{startSession}$ and $t_{k+1} = t_{endSession}$.

6.4.1 Proof without Conjunctions

Considering only the first case, in which all segments correspond to an expression of form ${}^{I_m,A_m}p_m$, it holds for each g_m that the corresponding observation function v_m is `true` for the whole duration of g_m.

W.r.t. g_1, a node n_1 which represents ${}^{I_1,A_1}p_1$ is created in the time-in state at $t_{startSession}$, because its predecessor node is n_{-1}, which is valid in the positive phase of the very first evaluation step, and the observation function v_1 changes to `true` in this very step.

Since n_{-1} terminates at $t_{startSession}$, and $duration(g_1) \leq A_1$, no time-out event has occurred until t_2. Since $duration(g_1) \geq I_1$, the node has left the time-in state and is in a valid state, at last in the positive phase of the evaluation step corresponding to t_2, or possibly earlier.

Therefore a valid node n_1 exists at the end of g_1 which represents a partial interpretation consisting of this single segment.

For each subsequent segment g_m of the interpretation i we assume inductively that there exists a node n_{m-1} which is in the valid state in the positive phase of the evaluation step corresponding to t_m, i.e. at the end of the preceding segment.[10]

This node has entered the valid state at some time instant $t_p \leq t_m$.

Since v_m is `true` during the whole g_m, and it is `false` before the very first evaluation step, it must have changed to `true` at some time instant $t_v \leq t_m$.

Therefore a node n_m representing g_m must have entered the time-in state at the time instant $t_n = \mathsf{latest}(t_v, t_p)$.

[10] If t_m happens to correspond to an evaluation step, the node n_{m-1} may leave its valid state in the negative phase of this evaluation step. This does not affect the following considerations.

Since n_{m-1} is in a valid state at least until the positive phase of the evaluation step at t_m, and since $duration(g_m) \leq A_m$, no time-out for n_m has expired until the end of g_m.

Since further $duration(g_m) \geq I_m$, and since v_m is still true at the end of g_m, the representing node n_m has left the time-in state and is still in a valid state at the end of g_m.

Therefore at the end of each segment a valid node exists which represents the partial specification (i.e. the prefix of i) up to and including this segment.

This is especially true for the last segment g_k. Because S_L is a linear specification derived from the *complete* SpecUT, this fact is recognized by the algorithm by $\triangle_\triangle$ being member of the set of subsequent expressions of n_k. Therefore this node is not only valid, but also contained in $finalNodes(GState.top)$ and a `pass` verdict is generated when calling $iFinalize()$.[11]

6.4.2 Proof including Conjunctions

Let the expression corresponding to g_m be of form $\texttt{AND}\{\alpha_1, \ldots, \alpha_r\}$. Then the existence of the interpretation i of the whole trace D implies the existence of interpretations $i_1, \ldots, i_r$ of g_m.

The segment g_m is the sub-trace $D_{[t_m \ldots t_{m+1}]}$.

Therefore each concatenation $j_x = \langle t_1, \ldots, t_{m-1} \rangle \frown i_x$ is a partial interpretation of $D_{[t_{t_{startSession}} \ldots t_{m+1}]}$.

If none of the $\alpha_1, \ldots, \alpha_r$ does contain an `AND` expression, it follows from the result of the preceding paragraph that at the time instant t_{m+1} for each such α_x there exists a node m_x in the valid state representing j_x.

One of these nodes has entered the valid state as the last, and in this very same evaluation step an `ASol` node a has been created.

Since every m_x represents, among others, the partial interpretation j_x, and $m_x.seFirst$ and $m_x.seLast$ reflect the possible start times of the segments represented by the leading node of m_x, it holds that for all m_x that $m_x.seFirst \leq t_m \leq m_x.seLast$.

Since the corresponding values of a are the latest and earliest of these values, it holds that ...

$$a.aeFirst \leq t_m \leq a.aeLast \tag{6.74}$$

Let I_x/A_x be the value yielded when summing up the minimal/maximal duration requirements imposed on the specifications contained in α_x. Let $a.minSubSum$ be equal to the largest of all I_x, and $a.maxSubSum$ be equal to the smallest of all A_x.

Since the segment g_m fulfills all α_x, it follows that ...

$$a.minSubSum \leq duration(g_b) = t_{m+1} - t_m \leq maxSubSum \tag{6.75}$$

[11] If the last specification particle is of the form ${}^{I_k, A_k}p_0$, i.e. it represents an `ANY` expression from $\mathcal{L}S$, and the maximal duration of the test session is known in advance, or $A_k = \infty$, the `pass` verdict may have been returned already by a preceding evaluation step, as an "early verdict".

Any time-in request imposed on a will expire at $a.seFirst + a.minSubSum$. For this value it holds that ...

$$\begin{array}{lcll} a.aeFirst & \leq & t_m & (6.74) \\ a.minSubSum & \leq & t_{m+1} - t_m & (6.75) \\ \hline a.seFirst + a.minSubSum & \leq & t_{m_+1} & \end{array} \tag{6.76}$$

Any time-out request imposed on a will expire at $a.seLast + a.maxSubSum$. For this value it holds that ...

$$\begin{array}{lcll} t_m & \leq & a.aeLast & (6.74) \\ t_{m+1} - t_m & \leq & a.maxSubSum & (6.75) \\ \hline t_{m+1} & \leq & a.aeLast + a.maxSubSum & \end{array} \tag{6.77}$$

From (6.76) it follows that at the time instant t_{m+1} the min-timer for a has expired, causing the transition of a from the time-in to the valid state.

From (6.77) it follows that the max-timer has *not* expired, not causing a transition from the valid to the terminated state.

Since additionally all nodes $m_x \in a.solParts$ are still valid at t_{m+1}[12], the `ASol` node a is at the end of g_m in a valid state, representing the complete partial interpretation up to g_m.

So the requirement that none of the $\alpha_1, \ldots, \alpha_r$ may contain an `AND` expression can be dropped by induction, and the same results as in the preceding section hold for arbitrarily nested interpretations.

[12] At least in the positive phase of a possibly happening evaluation step, which is sufficient.

6.5 Correctness and Completeness of Final and Early Verdicts

6.5.1 Formal Semantics of Verdict Values

Let $verdict\,(s : \mathcal{L}\,S', D : \mathcal{R}_+, t : \mathbb{T}) : Verdicts$ be the verdict value delivered by the algorithm at the time instant t, while matching the trace data D w.r.t. the specification $SpecUT$.

The semantics of the verdict value have been described informally in section 3.2: The early verdict `pass` means, that *all* possible continuations of the known prefix of the test data will fulfill the specification; the early verdict `fail` means, that *no* possible continuation of the known prefix of the test data will fulfill the specification:

This can be formalized using the auxiliary definitions from section 3.3 as[13] ...

$$
\begin{aligned}
& t < t_{endSession} \bullet \\
& \quad verdict(s, D_{[t_{startSession} \ldots t]}, t) = \texttt{pass} \implies \forall\, D' \bullet D_{[t_{startSession} \ldots t]} \frown D' \in [[SpecUT]]^L \\
& \quad verdict(s, D_{[t_{startSession} \ldots t]}, t) = \texttt{fail} \implies \forall\, D'' \bullet D_{[t_{startSession} \ldots t]} \frown D'' \notin [[SpecUT]]^L
\end{aligned}
\tag{6.78}
$$

Note that the implication arrows are *not* invertible: There are rare cases of a trace definitely fulfilling or failing a specification, which are not recognized immediately by the algorithm. In these cases the verdict `inconc` is delivered, in spite of one of the consequences in formula (6.78) is already true.

The optimal real-time property of the algorithm would imply that, as long as `inconc` is delivered, the cases of passing and failure are indeed both still possible:

$$
\begin{aligned}
& t < t_{endSession} \bullet \\
& \quad verdict(s, D_{[t_{startSession} \ldots t]}, t) = \texttt{inconc} \implies \exists\, D' \bullet D_{[t_{startSession} \ldots t]} \frown D' \in [[SpecUT]]^L \\
& \qquad\qquad\qquad\qquad\qquad\qquad \wedge \quad \exists\, D'' \bullet D_{[t_{startSession} \ldots t]} \frown D'' \notin [[SpecUT]]^L
\end{aligned}
\tag{6.79}
$$

This property holds for most cases of specifications, but not in general. The verdict `inconc` can also indicate that the algorithm is just not yet able do decide. This case is discussed in section 6.5.3 below.

The final verdict, which can only take the values `pass` and `fail`, is defined by ...

$$
verdict(s, D_{[t_{startSession} \ldots t_{endSession}]}, t_{endSession}) = \texttt{pass} \iff D \in [[SpecUT]]^L
\tag{6.80}
$$

The following section demonstrates that the properties (6.78) and (6.80) hold.

[13] Of course D' has to be "long enough", so that the concatenation $D_{[t_{startSession} \ldots t]} \frown D'$ yields a complete (finite or infinite) system trace. This trivial requirement has not been included in the formalization for the sake of readability.

6.5.2 Correctness and Completeness of Verdicts

The **final** verdict is derived from the state of the top level `OrGr` object after a test session (of finite duration) has ended at the time instant $t_{endSession}$.

For this purpose the function $iFinalize()^{(5.10)}$ calls $execute\_minMax()^{(5.18)}$ for a last time. This function executes all timer expirations which are still pending earlier than $t_{endSession}$, and all time-in requests which expire exactly at $t_{endSession}$. Consequently, if an evaluation step coincides with $t_{endSession}$, its positive phase is still executed.

Changes of observation functions must not be regarded in this final evaluation step, because their newly take values would correspond to some future time interval (cf. section 4.7.7), which is not longer a sub-interval of the test session.

If after this a final node, i.e. a node representing a partial interpretation which corresponds to a the *complete* SpecUT, is contained in *GState.top*, then it follows from (6.27) that the complete trace fulfills a linear specification derived from SpecUT. Therefore it fulfills SpecUT, and `pass` verdict must be delivered as the final verdict.

Contrarily, the fulfillment of SpecUT by D always implies the existence of an interpretation of D w.r.t. at least one linear specification derivable from SpecUT. Since this implies the existence of a valid node, as shown in 6.4, a `fail` verdict must be delivered if no such node exists.

This is exactly what function $deriveVerdict\_final()^{(5.12)}$ does.

The calculation of **early** verdicts by the function $deriveVerdict()^{(5.11)}$ anticipates this outcome of the final verict :

If during the execution no single node exists in *GState.top*, a top level final valid node needed for a final `pass` verdict can nevermore be created, since the creation of a node requires the existence of another valid node as its predecessor, cf. figure 4.3.

So whenever *GState.top* becomes empty because of the deletion of the last node contained therein, an early `fail` verdict is returned by $iNotify().^{(5.9)(5.11)}$, cf. the description in section 4.7.11.

Contrarily, whenever a final valid[14] `Prime` node exists in *GState.top* which (1) corresponds to the observation function v_0 (which is always `true`), and (2) the time-out expiration of which is known to happen after the end of the test session, then an early verdict of `pass` is delivered by $iNotify()$, because this node will survive until the very last evaluation step, and thus would in all cases cause a final `pass`verdict.

[14]This mechanism could be easily extended to treat final nodes p_0 in the time-in state as if they were in the valid state, if the time-in request is known to expire before $t_{endSession}$. This is not done in the current implementation for technical reasons.

6.5.3 Possible Early Verdicts not Recognized by the Algorithm

In the current version of the algorithm, the duration requirements imposed on an expression which is one of the linearizations of a chop sequence containing disjunctions is calculated dynamically(5.42).

Consider the following specification, given in front-end notation:

$$\alpha \;=\; \texttt{OR}\{\texttt{MIN}\,1\,p_0,\ \texttt{MIN}\,2\,p_1\}\ \underline{;}\ \texttt{OR}\{\texttt{MIN}\,4\,p_2,\ \texttt{MIN}\,8\,p_2\}$$

The different combinations which produce linear specifications and can appear in partial interpretations will have four different minimal duration requirements, symbolically written as ...

$$\texttt{OR}\{\texttt{MIN}\,5\,\alpha_1,\ \texttt{MIN}\,6\,\alpha_2,\ \texttt{MIN}\,9\,\alpha_3,\ \texttt{MIN}\,10\,\alpha_4\}$$

Now consider a conjunction like ...

$$\beta \;=\; \texttt{AND}\ \{\alpha,\ \texttt{MAX}\,9\,p_{10}\}$$

In the context of β a final node n_4 representing a partial interpretation w.r.t. the variant α_4 can obviously never be used to create an `ASol` node, because its duration requirement conflicts with the only specification contained in a parallel `OrGr`.

Therefore, as soon as the `OrGr` representing α in the context of β only contains node objects which lead to a final node of type n_4, the `ATst` node representing β could be discarded, which under appropriate circumstances could recursively lead to an early `fail` verdict.

But this type of failing a sub-specification is not recognized by the algorithm in the "early" way, — the duration requirements of parallel final nodes are compared not before the algorithm tries to combine them for creating a new `ASol` node, and even then the general impossibility to find any combination at all due to dynamic duratio requirements, is not recognized.

The possible general solution is to re-write the specification expression accordingly in the preparatory step which translates from $\mathcal{L}\,S$ to $\mathcal{L}\,S'$. This has been refrained from in the current implementation of the tool because of combinatorial explosion: the duration requirements given above in the definition of α can be read as just symbolic representations of duration requirements which are in turn dynamically defined by arbitrarily deep nested `AND/OR` expressions.

Nevertheless, if urgently required by an industrial application context of the tool, the performance of the algorithm could be improved by heuristic methods for detecting at least some of these cases at run-time.

6.6 Termination

Each single evaluation step of the kernel algorithm terminates. This follows from the termination of its positive and its negative phase, which can be shown separately:

- At the beginning of each system run there exists one single (pseudo-)node n_{-1}.
- Each event in the positive phase of an evaluation step, i.e. the expiration of a min-timer or the becoming-true of an observation function, causes — beside the change of the status of one or more node-objects — the creation of new nodes in the testing state.
  The number of the newly created nodes nodes related to a certain, currently valid node n as their predecessor, is limited by the cardinality of the set of subsequent expressions of n, as defined in section 4.7.5. Since all these expressions except `REPst` α are finite, this cardinality is finite, too.
  Additionally, all expressions of type `REPst` α can lead to new nodes maximally once in each evaluation step for a certain node serving as predecessor, since the algorithm includes active live-lock prevention(5.33). Therefore their installation does also cause only a finite set of new node objects.
- A certain node n entering its valid state may lead to the creation of new `ASol`-nodes, if n is a final node as defined in section 4.7.8.
  These newly created nodes are one `ASol` node for each possible combination of final nodes, one from each `OrGr` which represents a sub-expressions of the same `AND` expression in which the specification particle of n is contained.
  Since `AND`-expressions in $\mathcal{L}S$ are finite, the number of newly created `ASol` node objects is finite, too.

So in each positive phase the set of nodes grows only by a final number.

In the negative phase of each evaluation step the expirations of max-timers and the reaction to the becoming-false of observation functions do happen. These reactions can only decrease the number of nodes, which is a finite process, too.

Therefore each single evaluation step terminates. Consequently each call to an interface function of the kernel algorithm terminates.

6.7 Nodes in the Terminated State may be Deleted !

In the context of the propagation of information from a terminating nodes to all of its successors, as described at the end of section 6.3.4, it is of central importance for the efficiency of the algorithm that as soon as a node m goes to the terminated state, no further information on the further behaviour of *its predecessors* will be subsequently relevant for the semantics of any of its successors.

In other words: no communication between nodes is needed which "crosses" any node m which is in the terminated state. This allows in the implementation to simply "`mfree()`" the node m, thereby forgetting all structural information not only about m, but also about all of its predecessors.

Since the recursive calls of $LNode\_SEL\_lowers()^{(5.51)}$ and $ASol\_subSEL\_lowers()^{(5.52)}$ are the only event which is propagated forward through an existing tree of nodes, this can be shown as follows:

Let n_0 be the leading node of m and n_k be an immediate successor of $m = n_{k-1} = n_k.predec$ in the same node chain.

As soon as n_{k-1} enters the terminated state at time instant t_k, the value of $n_{k,t_k}.eLast$ is set to t_k, and $n_{k,t_k}.seLast$ is set to $T = \mathsf{earliest}(n_{k,t_k}.eLast - n_k.sumMinPreds, n_{k-1,t_k}.seLast)$, as defined by the parameter passed from $RNode\_becomesFixed()^{(5.49)}$ when calling $LNode\_SEL\_lowers()^{(5.51)}$.

The only reason for $n_k.seLast$ to be lowered further would be a lowering at some later time instant $t_x \geq t_k$ of some $n_x.seLast$ with n_x being a transitive predecessor of n_k in the same node chain.

This could have two different reasons:

Either n_x is a `Prime` node the predecessor of which terminates. In this case $n_x.seLast$ is set to $t_x - n_x.subMinPreds$.

But in this case it holds that $n_x.sumMinPreds \leq n_k.sumMinPreds$, so that $t_x - (n_x.sumMinPreds) \geq t_k - (n_k.sumMinPreds)$, so that the further lowering of $n_x.seLast$ does not cause a change of $n_k.seLast$, i.e. the value $.seLast$ of the immediate successor of the terminated node m, and consequently of none of its further transitive successors.

In the other case n_x is an `ASol` node, lowering its $.aeLast$ to a value $newAEL_x$, because of the lowering of $.seLast$ of some $n_S \in n_x.solParts$.

In this case it holds that

$$\begin{array}{lcll}
newAEL_x & \geq & t_x - n_x.minSubSum & (6.67) \\
\hline
newAEL_x - n_x.sumMinPreds & \geq & t_x - n_x.minSubSum - n_x.sumMinPreds & \\
n_k.sumMinPreds & \geq & n_x.minSubSum + n_x.sumMinPreds & (6.44) \\
\hline
newAEL_x - n_x.sumMinPreds & \geq & t_x - n_k.sumMinPreds & \\
t_x & \geq & t_k & \\
\hline
newAEL_x - n_x.sumMinPreds & \geq & t_k - n_k.sumMinPreds &
\end{array} \tag{6.81}$$

So this event neither has any effect on $n_k.seLast$, and all nodes in the terminated state can be discarded completely.

Chapter 7

Related Work

The areas of research and development where to find related work are those of *duration calculus*, *temporal logics* and *constraint resolution*.

The **duration calculus** (*DC*, cf. [3]) is the appropriate theoretical framework, into which the algorithm presented herein could be embedded. The basic setting is identical to fundament of the semantics of our specification language: the DC is a logic, the models of which are collections of functions from time to the set of Boolean values.

The full-scale DC turned soon out to be undecidable in the sense of mathematical logic [2]. Several sub-sets have been defined, subjected to mathematical research, and employed in model checking and theorem proofing [1].

Works towards execution are rare [7], [4]. Interestingly, the author of [7] also stresses that the finite variability of the input data is the key pre-requisite for executability, cf. page 4 above.

A small sub-set of DC could have been used as a frame-work for the formulation of the properties and proofs in this work. Due to significant formal over-head and only small benefits this has been refrained from.

Theories and tools from the different varieties of **temporal logics** (*TL*, cf. [11], TLA [9], ITL [15], TRIO [8] *etc.*) are a broad field of academic research, and partly already used in the industrial context, e.g. in circuit design and verification.

The first major difference between our approach and all approaches from this field (except [8]) is, that durations are not first order residents, but have to be modeled by a certain number of single, subsequent and identically defined "states". This does not only require a preparatory analysis on the bandwidth of the data, for defining a mapping from real-time intervals to these states. It also leads to an explosion of states if applied to multiple-clock data, because the real-time distance represented by a single state must correspond to the greatest common divider of the distances of all possible critical time instances.

Just contrarily, our approach can be applied immediately to arbitrarily defined domains representing time, including those with a *dense* structure. The implementation is limited only technically, but not semantically, by the precision of the employed infra-structure.

The second major difference lies in the concrete technologies of application and implementation:

On the one hand, TL formula are fundamental to the various techniques of model checking [6], and are subject to theorem proving [10], with all the well-known restrictions to these technologies.

On the other hand, there are several activities of genuine and real-time capable TL tool construction [14], [5], [17]. The general strategy in this area is to derive an automaton, which is able to monitor a data trace using constant space and time. This is not feasible for dense time domains, as in our case and in the case of [8]. Therefore, in the algorithm presented herein, an equivalent to this automaton exists only virtually, and is extended and dismantled dynamically on demand.

Significant similarity with our approach can be found in [16]. This recent work presents a method for generating an automaton which matches a sequence of real-time events against a regular expression.

All these tools are superior to our approach w.r.t. the constant space and time property, and because TL supports *negation*, which cannot be integrated in our specification language as a free constructor. They are inferior because they cannot deal naturally with durations.

From the viewpoint of **constraint resolution** (*CR*), our approach could be seen as a very specialized form of *incremental* CR.

The arithmetic components of the specifications processed by our algorithm are only very primitive linear constraints, so that this work is not a contribution to CR in the narrow sense.

The complications come from the execution context: Several initially independent constraints are processed in parallel. The detection of a certain solution (i.e. "becoming valid of a node object") leads to decisions which are determined by Boolean logic. These decisions, in turn, lead to a dynamic creation of new constraints by combining data from different solutions.

The severe problems involved in incremental CR in the general case do not apply to our algorithm: All decisions concerning the solution tactics are uniquely determined by the sequential structure of the SpecUT and the observed behaviour of the SUT.

Since no direct predecessor has been found published, the author is currently (November 2003) applying for a European patent on the algorithm presented herein.

Index

Bibliography

[1] Alf-Christian Achilles. The collection of computer science bibliographies — *query* =`duration calculus`. In `http://liin.ira.uk.de`, 2003.

[2] Zhou ChaoChen, Michael R. Hansen, and Peter Seftoft. Decidability and undecidability results for duration calculus. In *Symposium on Theoretical Aspects of Computer Science STACS'93*, volume 665 of *LNCS*. Springer Verlag, Berlin, Heidelberg, New York, 1993.

[3] Zhou ChaoChen, Tony Hoare, and Anders P. Ravn. A calculus of durations. In *Information Processing Letters*, 1991.

[4] Nathalie Chetcuti-Sperandio. Tableau-based automated deduction for duration calculus. In Uwe Egly and Christian G. Fermüller, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 2381 of *Lecture Notes in Computer Science*, pages 53–69. Springer-Verlag, July 30–August 1 2002.

[5] D.Giannakopolou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Proceedings of the International Conference on Automated Software Engineering, ASE'01*. IEEE, 2001.

[6] D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*. Springer Verlag, Berlin, Heidelberg, New York, 2000.

[7] Martin Fränzle. Synthesizing controllers from duration calculus. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '96)*, volume 1135, pages 168–187, 1996.

[8] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO, a logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 1990.

[9] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 17(3), 1995.

[10] Z. Manna and N. Bjoerner et.al. An update on StEP: deductive-algorithmic verification of reactive systems. In *Tool Support for System Specification, Development and Verification*, LNCS. Springer Verlag, Berlin, Heidelberg, New York, 1998.

[11] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Specification.* Springer Verlag, Berlin, Heidelberg, New York, 1991.

[12] The MathWorks. *Matlab, the Language of Technical Computing — Using Matlab.* Natick, MA, USA, November 2000.

[13] The MathWorks. *Simulink, Dynamic System Simulation for Matlab — Using Simulink.* Natick, MA, USA, November 2000.

[14] Ben Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1986.

[15] Ben Moszkowski and Zohar Manna. Reasoning in interval temporal logic. In *Proceedings of the Workshop on Logic of Programs*, LNCS, pages 371–382. Springer Verlag, Berlin, Heidelberg, New York, 1983.

[16] Koushik Sen and Griore Rosu. Generating optimal monitors for extended regular expressions. In *Proceedings of the RV'03 — Third Workshop on Runtime Verification.* Elsevier, to appear.

[17] Koushik Sen, Griore Rosu, and Gul Agha. Generating optimal linear temporal logic monitors by coinduction. In *Proceedings of the ASIAN'03*, LNCS. Springer Verlag, Berlin, Heidelberg, New York, 2003.

[18] C.E Shannon. The mathematical theory of communication. *Bell Systems, Technical Journal*, 27, 1948.

[19] J.M. Spivey. *The Z Notation — A Reference Manual.* Series in Computer Science. Prentice Hall International, 2nd edition, 1992.

Acknowledgements

The author owes special thanks to former and current members of the ÜBB group at the Technische Universität Berlin:

First of all to Wolfgang Grieskamp, now at microsoft research, who initiated the industrial cooperation and who took part in the first steps of front-end language design.

To Thomas Nitsche, Baltasar Trancón y Widemann and Jacob Wieland, for many fruitful discussions which brought my work forward significantly, and to all members (and the heads) of the SWT and the ÜBB group, for the friendly, co-operative and nevertheless challenging atmosphere.

To the colleagues at DaimlerChrysler/FT3/SM for valuable suggestions concerning the tool design, and, last not least, to the head of our group, Peter Pepper. Without his support this work would not exist.

Appendix A

Notational Conventions and Global Abbreviations

S	Front-end specification language.
S'	Middle-end language. Sentences from S' serve as input to the kernel algorithm.
S''	Back-end language. Sentences from S'' describe prefix of a partial interpretation which has been accepted by a valid node.
$\mathcal{L}\, s$	The language generated by a non-terminal s, i.e. the collection of final sentences derivable from s.
${}^{a,i}p_k$, $\underline{;}$, AND, OR, OPT, REP, REPst, MIN, MAX	Operators of S and S'.
$\triangle_{\triangle}$	Reserved terminal symbol from S', used for detecting that a sub-expression of an AND/OR expression is completely fulfilled.
$\mathcal{R}_+$	The set of all non-empty traces w.r.t. a certain collection of atomic predicates.
$\mathcal{R}$	The set $\mathcal{R}$, plus additionally the empty trace.
$\_ \frown \_$	Concatenation function for traces of type $\mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}$.
$\_ \frown \_$	Concatenation of interpretations of type $\mathbf{seq}\,\mathbb{T} \times \mathbf{seq}\,\mathbb{T} \rightarrow \mathbf{seq}\,\mathbb{T}$.
$[[e]]^L$	A function from $\mathcal{L}\,S \ \ldots \mathcal{L}\,S''$ into $\mathbf{set\ of}\,\mathcal{R}$, giving the set of all traces which fulfill the specification expression e.
D	The trace data as produced by an SUT and the adaptive layer during one certain session.
$D_{[t_1 \ldots t_2]}$	The sub-trace of D extending from t_1 up to t_2.
p_k	Atomic predicate: a specification expression which expresses that the corresponding observation function v_k stays continuously `true`.
${}^{i,a}p_k$	Specification expression from the middle-end language S'. It corresponds to the front-end notation `MIN` i `MAX`a p, and is fulfilled by all sub-traces which fulfill p_k and have a duration d which fulfills $i \leq d \leq a$.
v_k	The observation function indicated by the predicates p_k and ${}^{i,a}p_k$.
$f(\!\lvert\sigma\rvert\!)$	The map operator. $f(\!\lvert\sigma\rvert\!)$ is the collection of results yielded by applying the function f to each member of the collection σ.
$\mu\ \sigma$	The μ operator selects the one and only member of the collection σ, iff this is of cardinality one(1). Otherwise it is undefined.
$r^{\sim}$	The inverse of the relation r.
$\mathsf{dom}\, f / \mathsf{ran}\, f$	Domain and range of a function or relation.
$\mathsf{lub}\, \sigma / \mathsf{glb}\, \sigma$	Least Upper Bound / Greatest Lower Bound of a collection of values, on which an order is defined.
$\mathsf{latest}(\sigma) / \mathsf{earliest}(\sigma)$	Functions delivering the minimal/maximal value contained in a collection σ of time instance values.

Appendix B

Technical Manual of the *MW*ATCH Tool

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

MMATCH Installation and Users Guide

Markus Lepper
— ÜBB/TU Berlin —

Contents

List of Figures

List of Tables

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 `:-)`

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 ...

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.

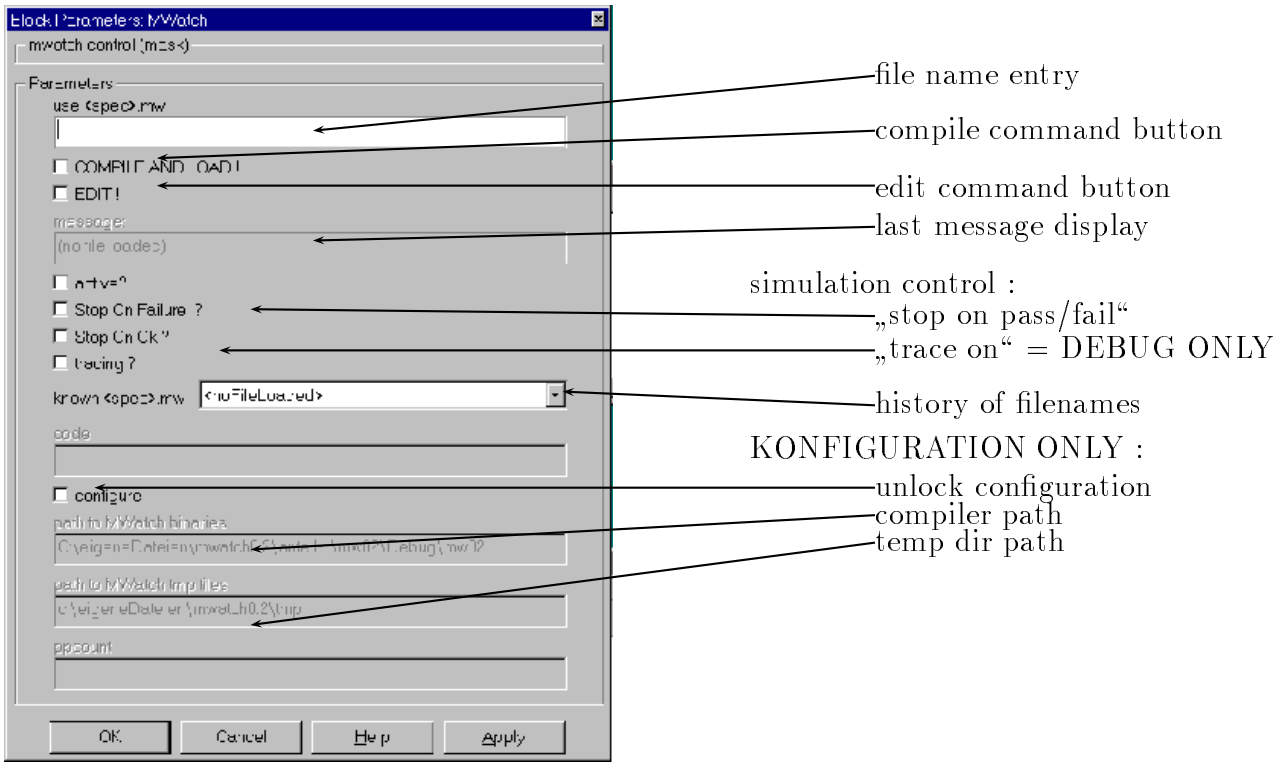
The check buttons below are *abused* as command buttons, because such are not foreseen in a simulink mask, e.g. clicking on these boxes will make a check mark appear and be deleted again immediately, after the command is executed.

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

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 MWATCH-blocks mask, and (**preliminary :**) in two dedicated output fields in the mask.

3.3 Resolving of file names – Windows NT version

All files for MWATCH (either specifications given to MWATCH-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 MWATCH API for Program Controlled Watchdog Installation

There is a small API providing MATLAB functions for installation, control and de-installation of MWATCH 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 MWATCH-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 MWATCH Console for Controlling Multiple Predicates from One File for One Subsystem

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

This is realized by automatic insertion and deletion of MWATCH blocks.

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

`mwInstall`	($\langle SubsystemName \rangle$, $\langle MWatchFileAndSect \rangle$, $\langle blockName \rangle$)
$\Rightarrow$	[$\langle ErrorCode \rangle$, $\langle ErrorText \rangle$, $\langle MwBlockPath \rangle$]
	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 $\langle blockName \rangle$ (a MATLAB array of length=0) is given, a fresh name is supplied automatically.
`mwRelink`	($\langle MwBlockPath \rangle$, $\langle MWatchFileName \rangle$)
$\Rightarrow$	[$\langle ErrorCode \rangle$, $\langle ErrorText \rangle$]
	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`	($\langle MwBlockPath \rangle$, $\langle Integer \rangle$, $\langle Integer \rangle$)
$\Rightarrow$	[$\langle ErrorCode \rangle$, $\langle ErrorText \rangle$]
	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`	($\langle MwBlockPath \rangle$)
$\Rightarrow$	[$\langle ErrorCode \rangle$, $\langle ErrorText \rangle$]
	Removes the MWATCH-block.
`mwGetverdict`	($\langle MwBlockPath \rangle$)
$\Rightarrow$	[$\langle ErrorCode \rangle$, $\langle ErrorText \rangle$, $\langle verdict \rangle$, $\langle verdictTime \rangle$]
	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

`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 rescanned 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`“ :

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*⟩
	|	⟨*mwatchText*⟩$^?$ ⟨*fileSection*⟩$^+$
⟨*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 MWATCH Language of Temporal Predicates

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

Each term of an MWATCH 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 MWATCH-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 MWATCH-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 MWATCH 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.)

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 "$\sim$" (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.

`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`	"(" ⟨*arithExpr*⟩ ")"
`cos tan asin acos atan atan2 sinh cosh`	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 MWATCH! 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

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 $\langle instPred \rangle$ 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

$$\texttt{SET}\ \langle testPoint \rangle\ \texttt{"="}\ (\langle arithExpr \rangle \mid \langle boolExpr \rangle)\ \texttt{";"}$$

where $\langle testPoint \rangle$ 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.

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 MWATCH Macro Facility

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

please note : that, as mentioned above, when evaluating an MWATCH-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 MWATCH-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.

- 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!?

Figure 5 The grammatical circles of the MWATCH language

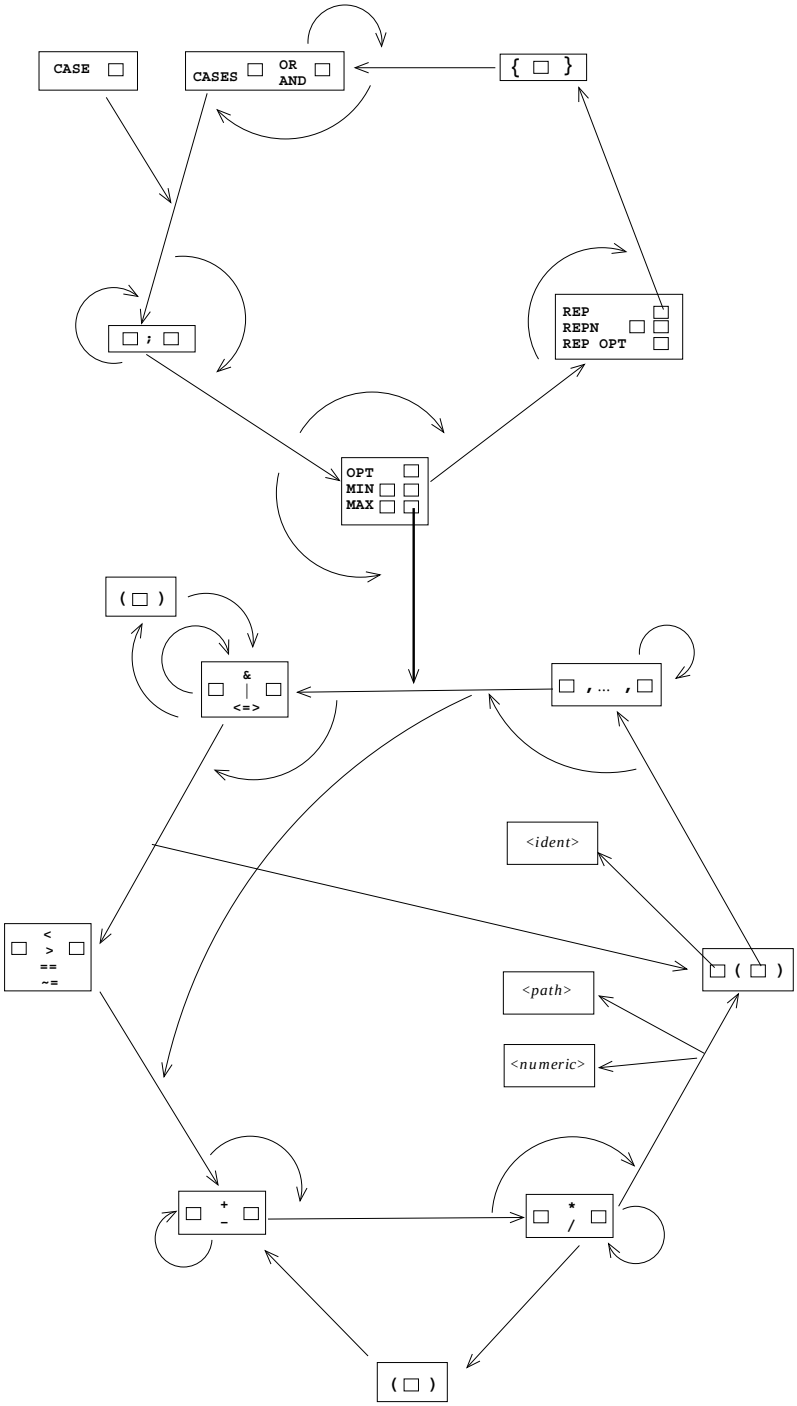


Figure 6 Schematic Grammar of the MWATCH Predicate Language.

```
⟨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 MWATCH 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⟩
```

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.

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]") ;
```

```
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
```

```
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

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`

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 MWATCH 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.

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 „$\sim$ `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 `kickdown`becoming 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 instantatneous 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.

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.

Appendix C

Tutorial On the Writing of Specifications

MMATCH Extended Tutorial

Markus Lepper
— ÜBB/TU Berlin —

Contents

List of Figures

List of Tables

1 Introduction to the MWATCH Principles

[0] The following text is an attempt of a small interactive tutorial, introducing several aspects of the MWATCH 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 MWATCH algorithm,
- the MWATCH tool,
- and the MWATCH 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 MWATCH 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 MWATCH *language*. The MWATCH 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 MWATCH 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 MWATCH 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 non-deterministic 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 MWATCH 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 MWATCH: 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 MWATCH 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“.

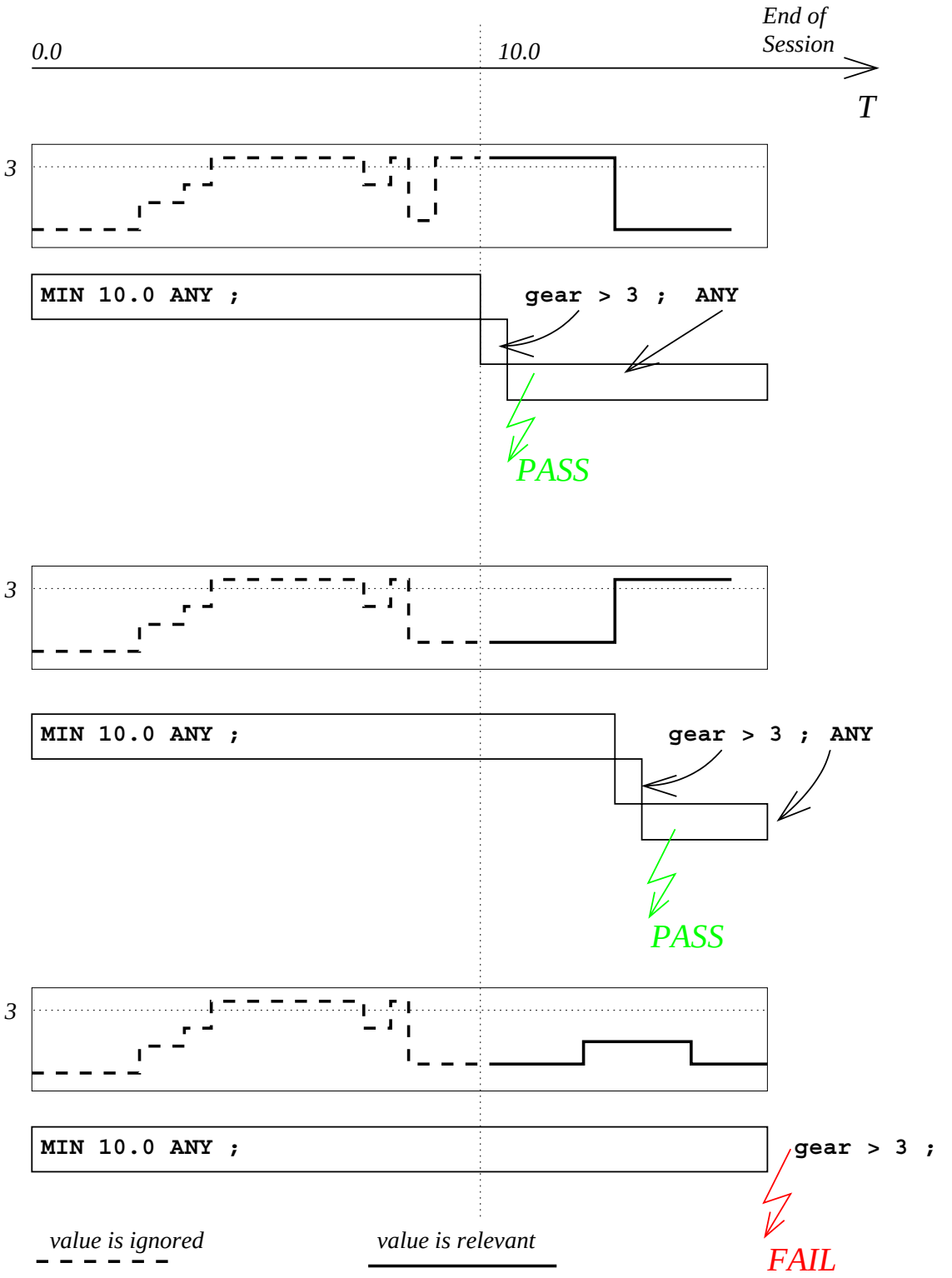
Or:

„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 :

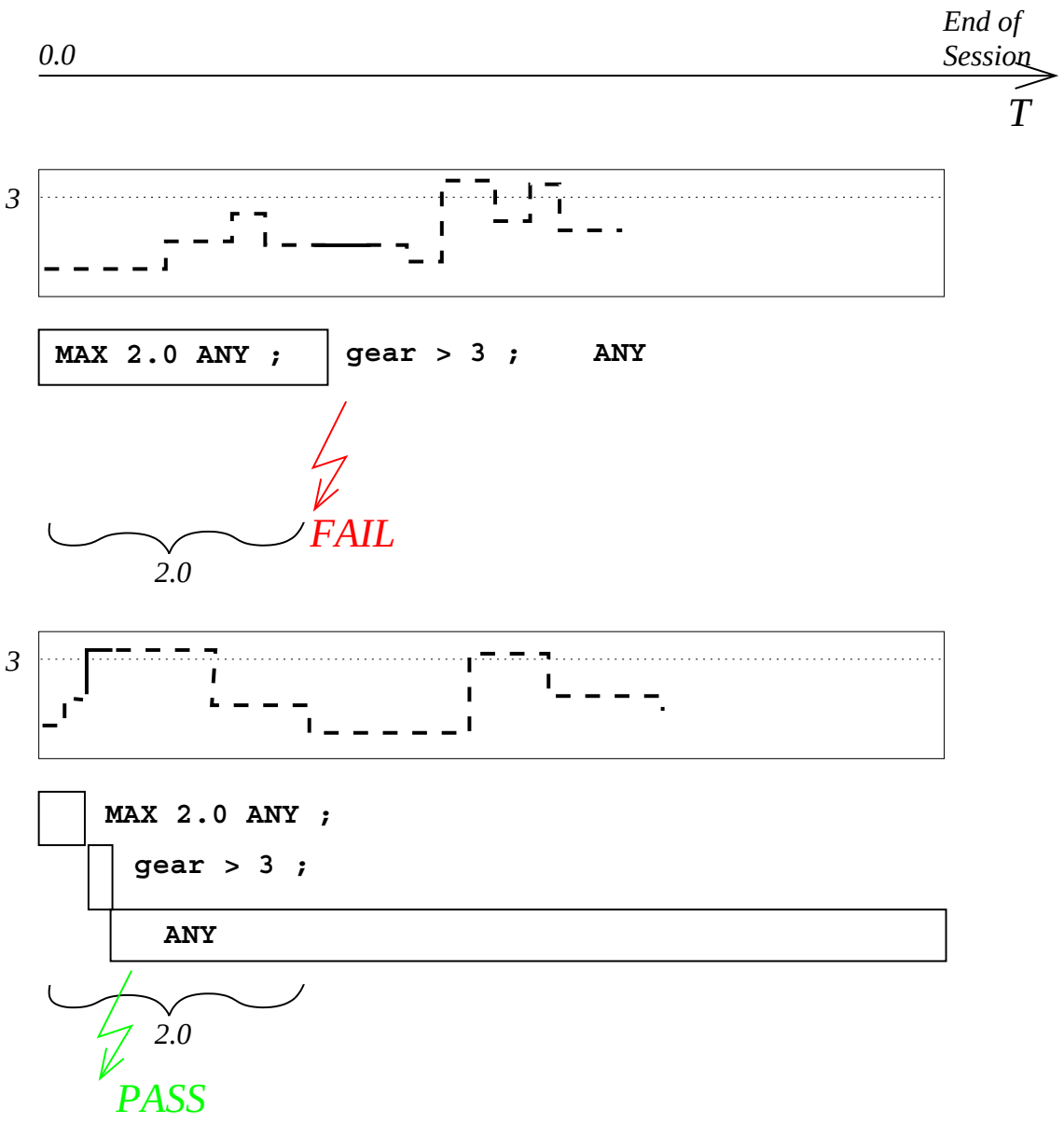
```
CASE MAX 2.0 ANY  ; shift_logic/gear > 3 ; ANY  EOF
```

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
```

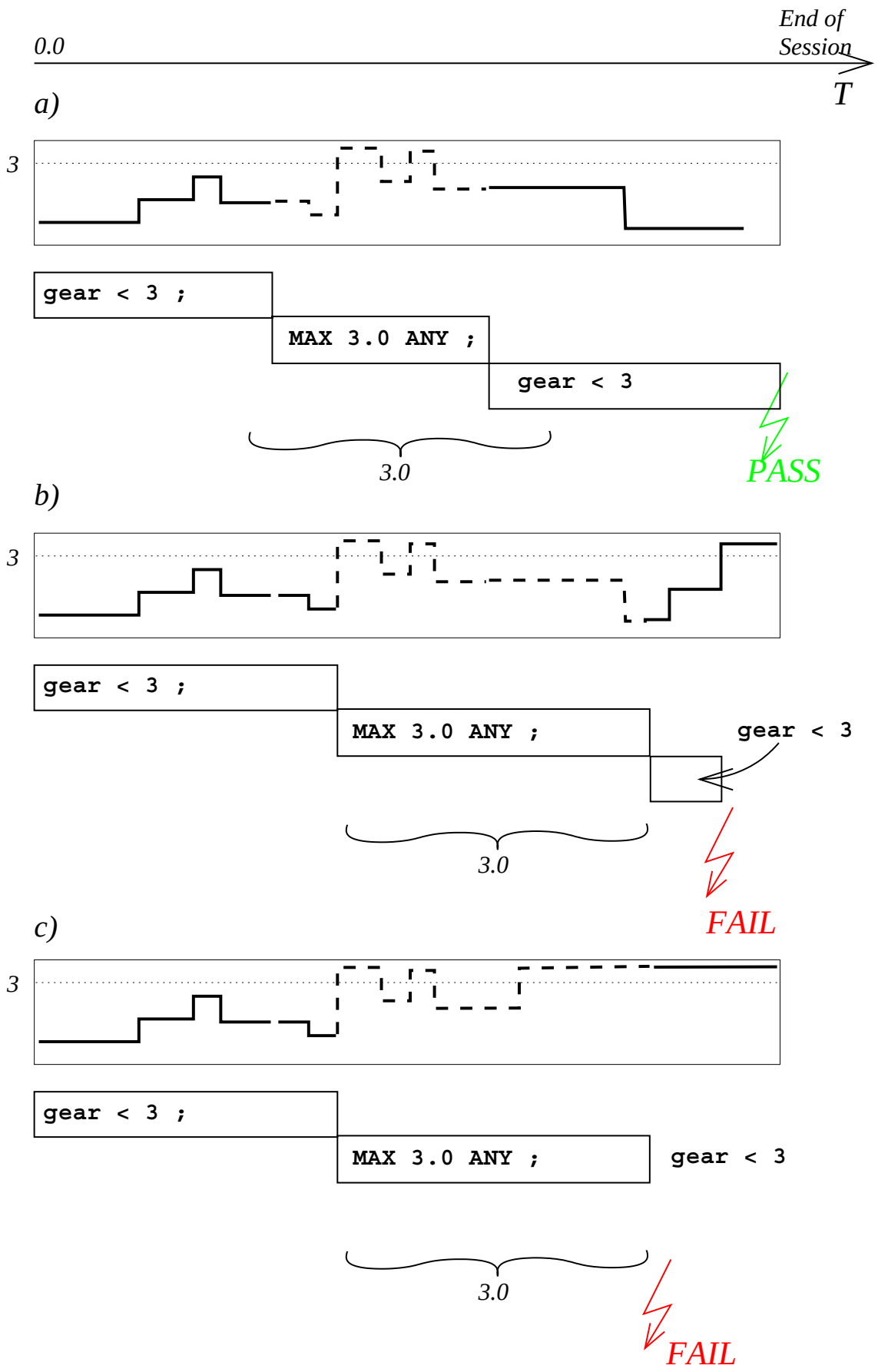
Since we claim for the gear being ≤ 3 in the first and last segment, only the middle segment would allow a gear larger than 3, because it does *not* impose this restriction.

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 $\leq$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 MWATCH 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 MWATCH 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 MWATCH-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 MWATCH! 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 MWATCH 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:

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

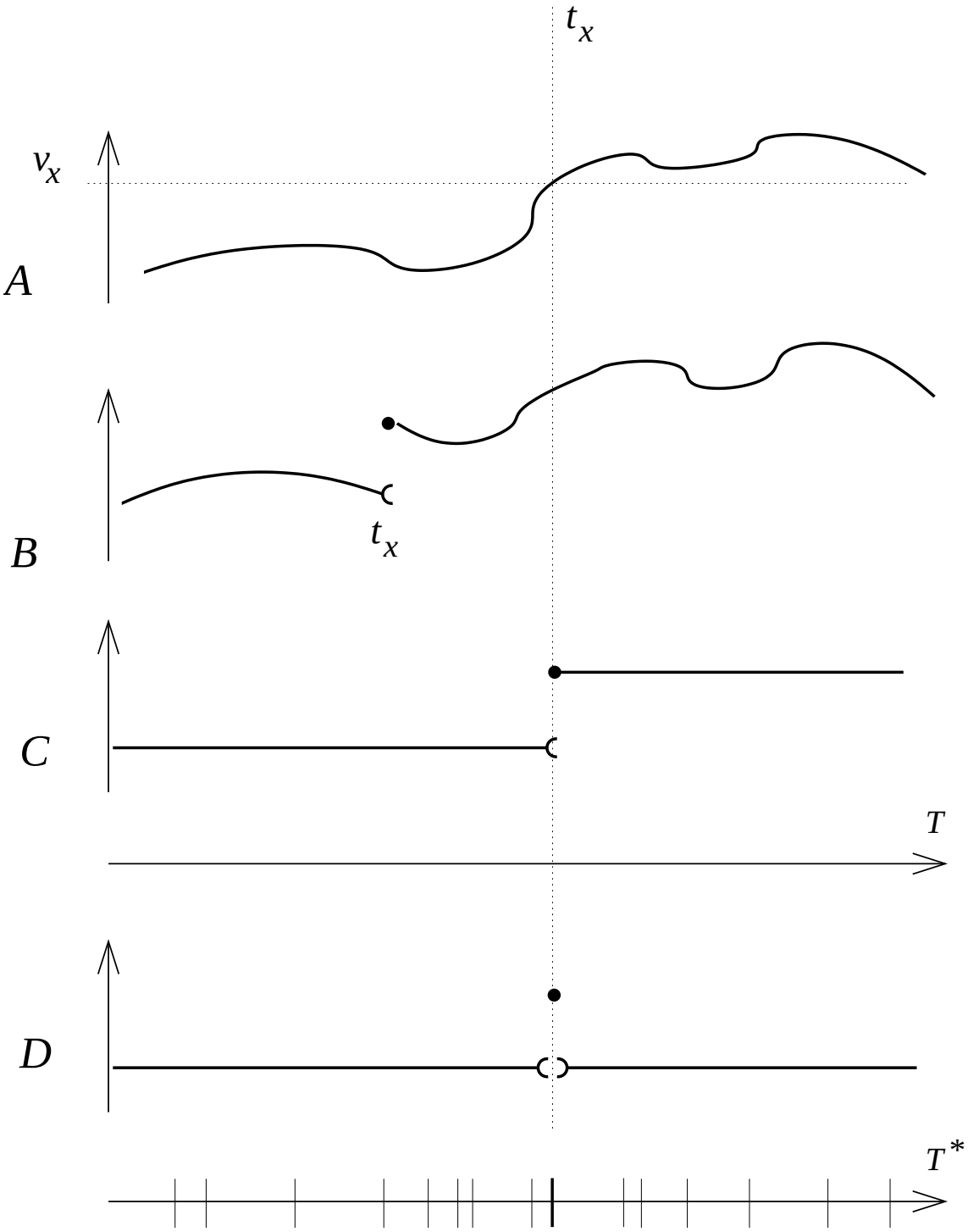
The signal `gear` in the preceding examples is such a „discrete“ signal.

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

LET D = memory (B) - 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 MWATCH evaluation: MWATCH 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

MWATCH.

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 MWATCH 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 MWATCH operates in the world of non-determinism, it hardly makes any sense to „feed back“ signals generated by MWATCH 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 MWATCH 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.