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*.

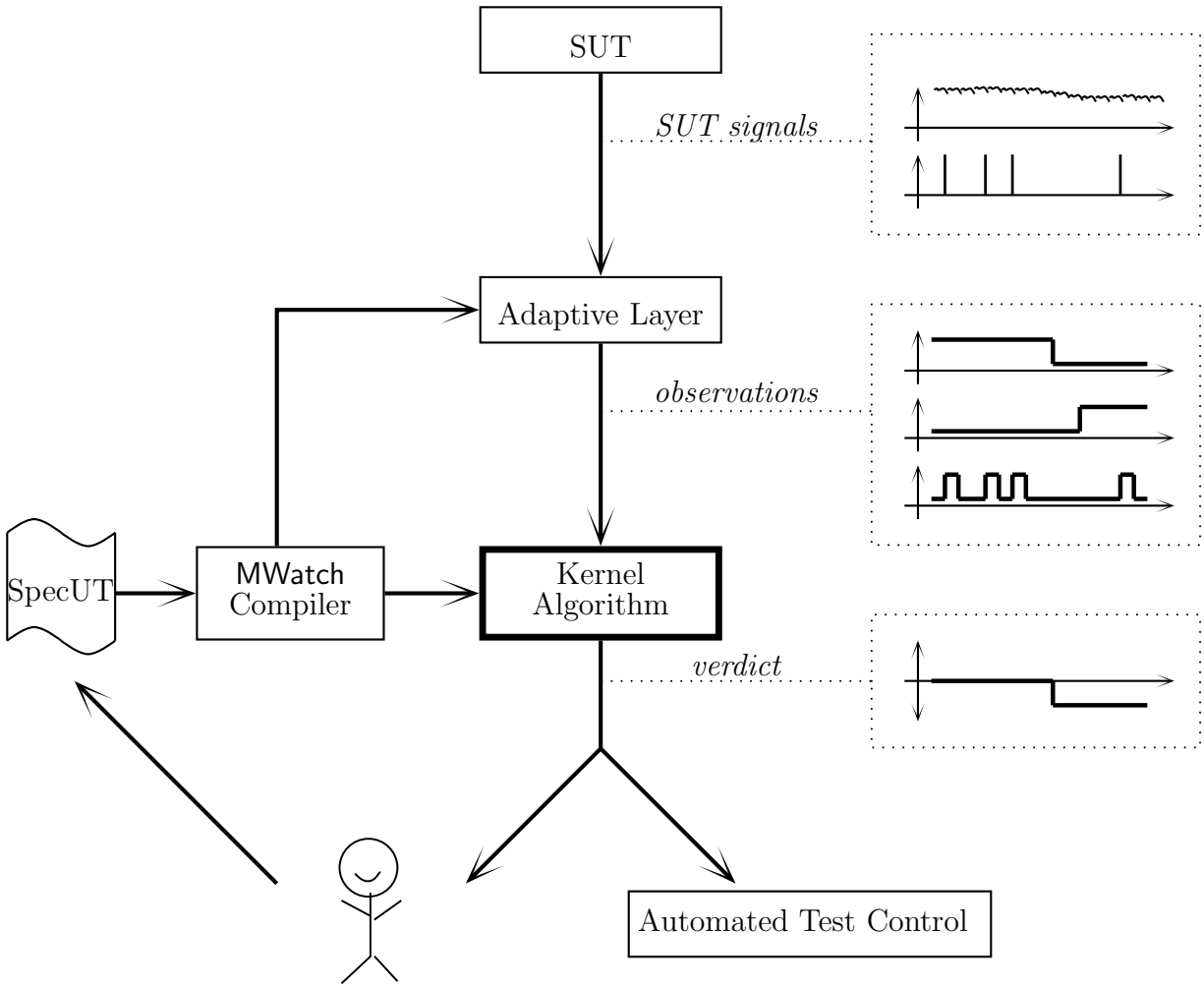
Analog signals may be either produced by analog/digital-converter hardware (ADCs), when observing a real physical system, or by a digital simulation model which delivers representations of analog values in discrete steps using e.g. an equation solver.

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 MWATCH 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 MWATCH 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 MWATCH 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

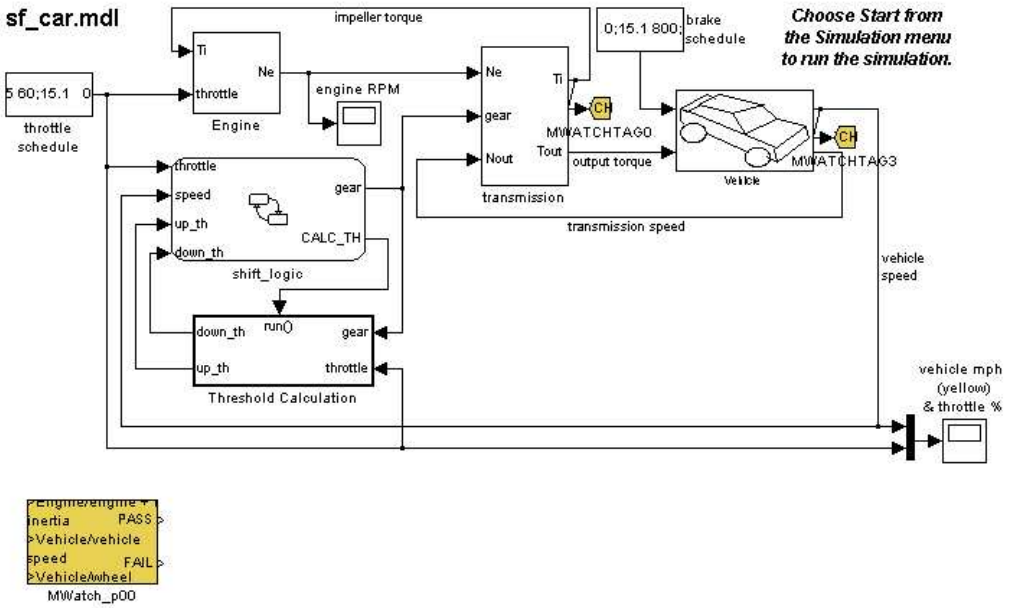
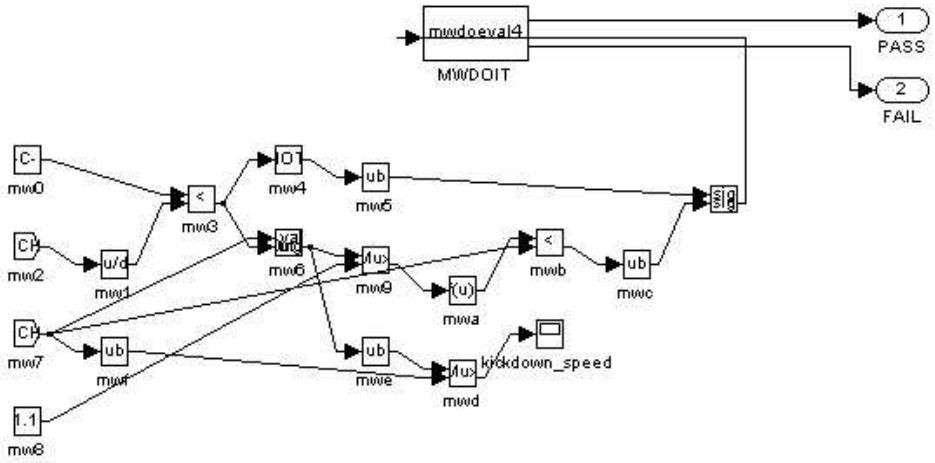


Figure 2.3 A Signal Processing Network Realizing an Adaptive Layer



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:

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

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 $\mathsf{conc} : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$ 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 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

$$
\begin{array}{lcl}
\mathit{Boolean} & = & \{\texttt{false}, \texttt{true}\} \\
\mathbb{T} & = & \mathbb{R}_{\geq 0.0} \\
\mathbb{D} & = & \mathbb{R}_{\geq 0.0}
\end{array}
$$

$$
\begin{array}{lcl}
P_+ & = & \{p_1, p_2, \ldots, p_{maxp}\} \\
P & = & P_+ \cup \{p_0\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{lb}(T : \textbf{set of } \mathbb{T}) : \textbf{set of } \mathbb{T} & = & \{t : \mathbb{T} \mid (\neg\ \exists\, t_2 \in T \bullet t_2 < t)\,\} \\
\mathsf{ub}(T : \textbf{set of } \mathbb{T}) : \textbf{set of } \mathbb{T} & = & \{t : \mathbb{T} \mid (\neg\ \exists\, t_2 \in T \bullet t_2 > t)\,\} \\
\mathsf{glb}(T : \textbf{set of } \mathbb{T}) : \mathbb{T} & = & \mu\{t \in \mathsf{lb}\ T \mid (\neg\ \exists\, t_2 \in \mathsf{lb}\ T \bullet t_2 > t)\,\} \\
\mathsf{lub}(T : \textbf{set of } \mathbb{T}) : \mathbb{T} & = & \mu\{t \in \mathsf{ub}\ T \mid (\neg\ \exists\, t_2 \in \mathsf{ub}\ T \bullet t_2 < t)\,\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{R}_+ & = & \{r : \mathbb{T} \rightarrow P \rightarrow \mathit{Boolean} \\
 & & \mid\ \mathsf{glb}\ (\mathsf{dom}\, r)\ <\ \mathsf{lub}\,(\mathsf{dom}\, r) \\
 & & \wedge\ \forall\, t\ \in\ \mathbb{T}\ \mid\ \mathsf{glb}\ (\mathsf{dom}\, r)\ \leq\ t <\ \mathsf{lub}\,(\mathsf{dom}\, r) \\
 & & \qquad\qquad \Longrightarrow\ t \in \mathsf{dom}\, r\ \ \wedge\ \ P = \mathsf{dom}(r\ t)\} \\
\mathcal{R} & = & \mathcal{R}_+ \cup \{\{\}\}
\end{array}
$$

$$
\mathsf{conc}(r_1 : \mathcal{R}, r_2 : \mathcal{R}) : \mathcal{R} \quad = \quad
\begin{cases}
\textbf{if } r_2 = \{\} \textbf{ then } & r_1 \\
\textbf{if } r_1 = \{\} \textbf{ then } & r_2 \\
\textbf{otherwise} & \\
\quad r_1\ \oplus\ (r_2 \circ (\lambda\, t\ \bullet\ t - \mathsf{glb}\ (\mathsf{dom}\, r_2) + \mathsf{lub}\ (\mathsf{dom}\, r_1))) &
\end{cases}
$$

$$
\begin{array}{l}
\mathsf{combine}\,(w_1 : \textbf{set of } \mathcal{R}, w_2 : \textbf{set of } \mathcal{R}) : \textbf{set of } \mathcal{R} \\
\qquad = \ \lambda\ x_1, x_2 \bullet \mathsf{conc}\,(x_1, x_2)\ (\!|w_1 \times w_2|\!)
\end{array}
$$

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'

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

This interface function returns an early verdict value, which is one of the values $\{\texttt{pass}, \texttt{fail}, \texttt{inconc}\}$, 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.

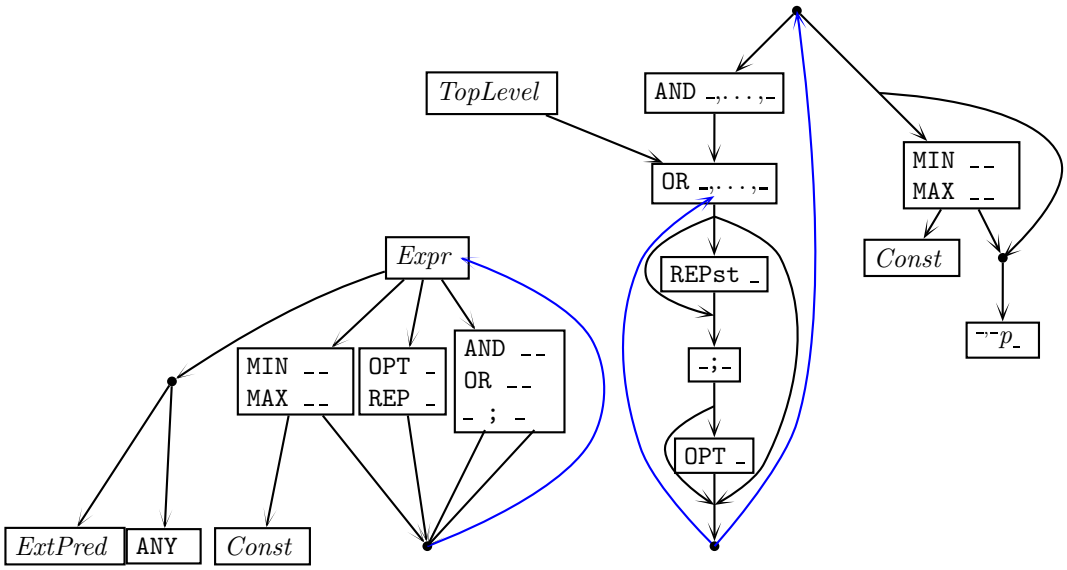
4.2 Normalization of Specification Terms

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 ...

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

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 $\alpha \,\underline{;}\, \neg p_k$ 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 $t_1 = \mathsf{glb}\,\mathsf{dom}\,D$ and $t_{k+1} = \mathsf{lub}\,\mathsf{dom}\,D$, 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 $\dashv p_k$.

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 subexpression 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

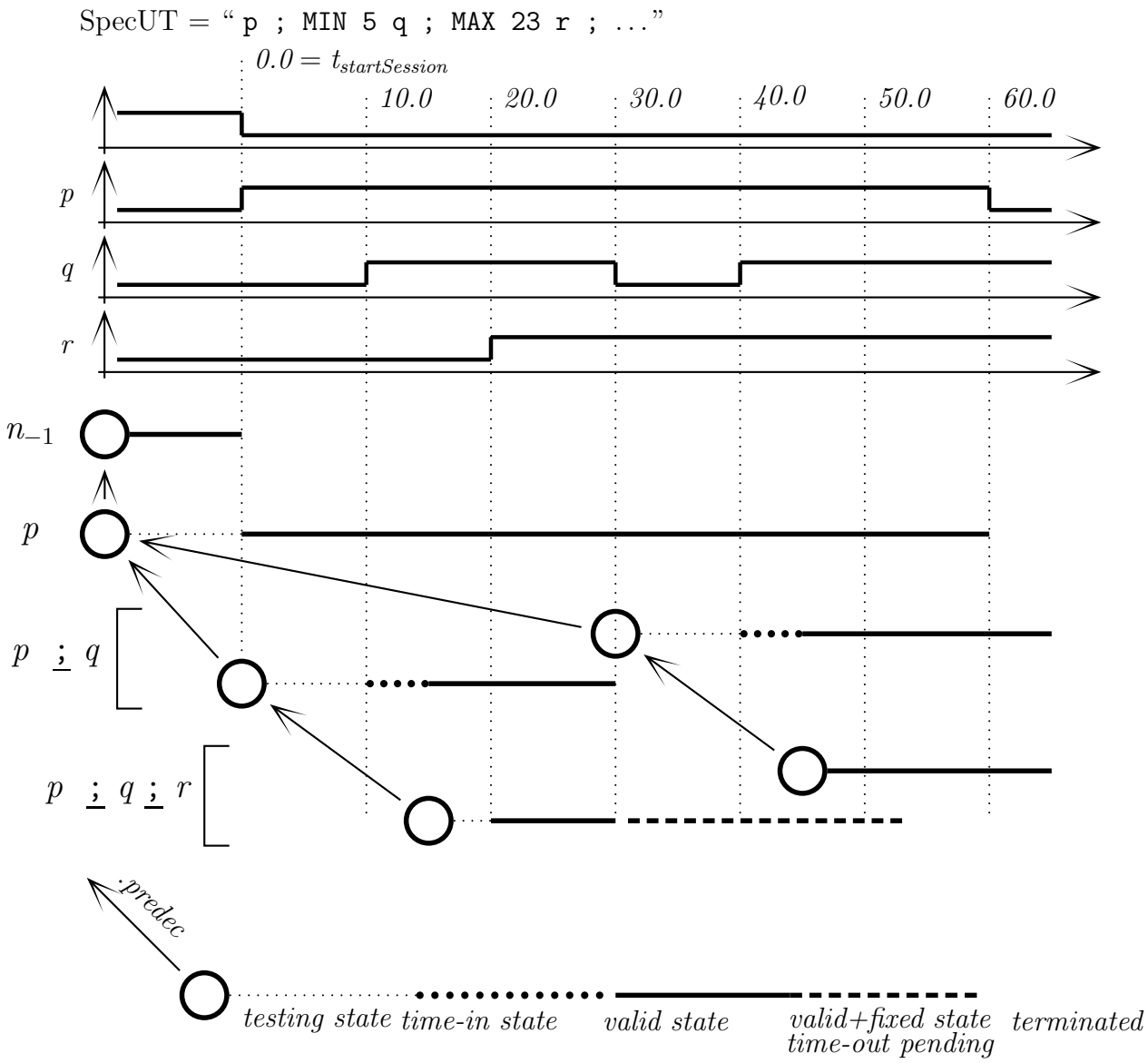
The predecessor function is realized in the algorithm by the node attribute $n.predec$.

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 $\texttt{--}p\_$ or of form $\texttt{AND}\{\ldots\}$.

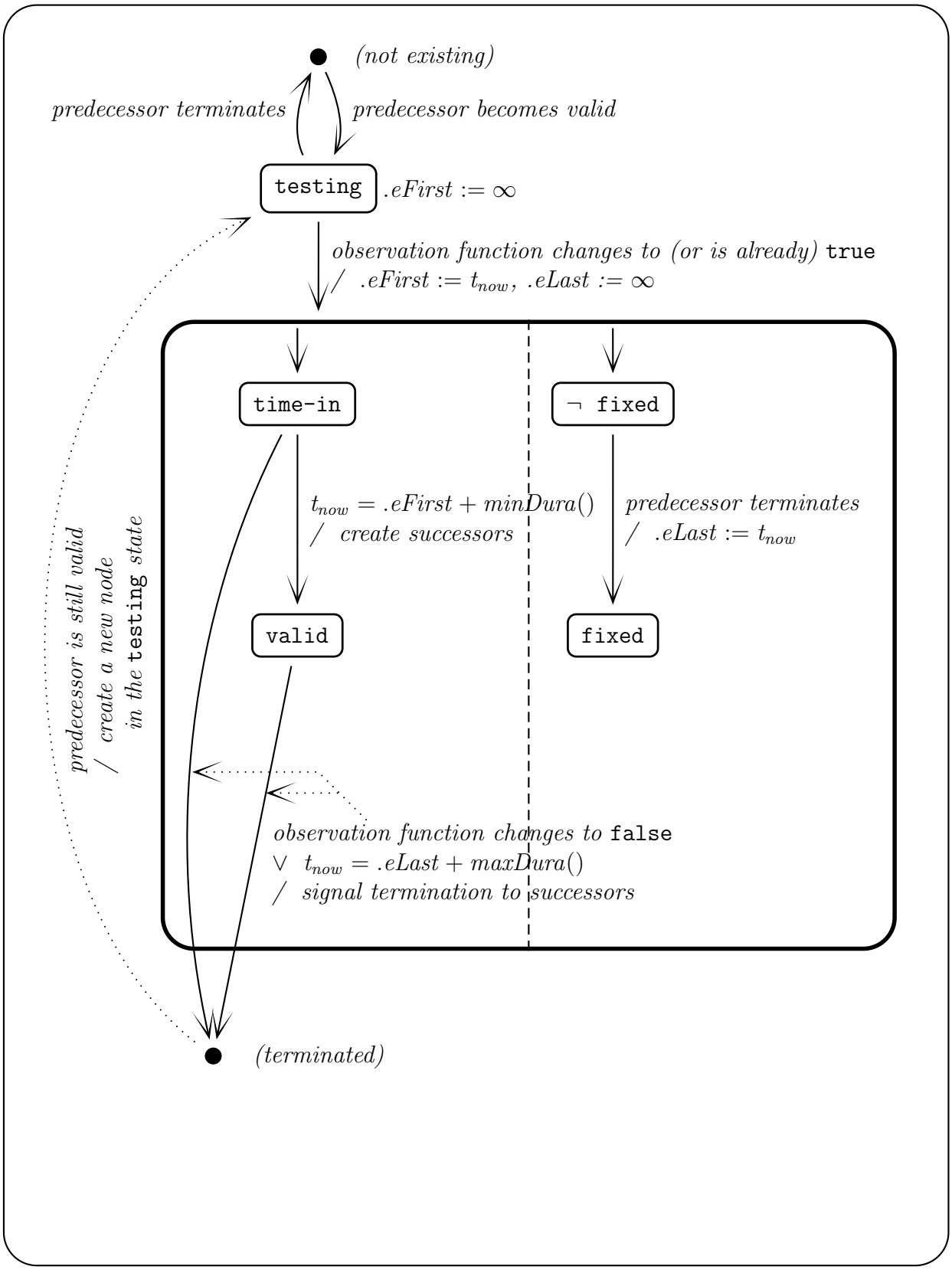
Therefore, in the same evaluation step in which a node becomes valid$^{(5.31)}$, the set of its subsequent expressions is calculated, and for each element e of this set a new node n_e is created. $^{(5.32)(5.33)(5.35)}$ The value of $n_e.expr$ is set to e, and the value of $n_e.predec$ is set to n, and 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 $\alpha \mathbin{;} \triangle_\triangle$. (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 $p_1 \underline{;}\, \texttt{MAX}\, d_2\, p_2 \underline{;}\, p_3$.

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 ...

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

... 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. $p_1 \underline{;}\, \texttt{MAX}\, 7\, p_2 \underline{;}\, p_3$ can start anywhere between 0.0 and 100.0, but the provisional interpretations w.r.t. $\texttt{MAX}\, 7\, p_2 \underline{;}\, p_3$ 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 MWATCH 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) ...

$$\texttt{AND}\,\{\ (p_1\ \underline{;}\ \ldots),\ \texttt{OR}\,\{\ (p_2\ \underline{;}\ \ldots), (p_3\ \underline{;}\ \ldots),\ \texttt{AND}\,\{\ (p_4\ \underline{;}\ \ldots), (p_5\ \underline{;}\ \ldots)\ \}\ \}\ \}$$

... 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

$$
\begin{array}{lcl}
\mathit{Verdicts} & = & \{\,\texttt{pass}, \texttt{fail}, \texttt{inconc}\,\} \\
\mathbb{D}_+ & = & \mathbb{D} \cup \{\infty\} \\
\mathbb{T}_+ & = & \mathbb{T} \cup \{\infty\}
\end{array}
\tag{5.1}
$$

$$
\begin{array}{lcl}
\mathit{Values}_+ & = & \{p_1, \ldots, p_{\mathit{GState.pCount}}\} \rightarrow \mathit{Boolean} \\
\mathit{Values} & = & \{p_0, p_1, \ldots, p_{\mathit{GState.pCount}}\} \rightarrow \mathit{Boolean} \\
\multicolumn{3}{l}{\mathit{internalize}(v : \mathit{Values}_+) : \mathit{Values} \;=\; v \cup (p_0 \mapsto \texttt{true})}
\end{array}
\tag{5.2}
$$

$$
\begin{array}{lcll}
\mathit{GState} & = & \textbf{struct}\,\{ & \mathit{nodes} : \quad \mathit{ObjSet} \\
 & & & \mathit{top} : \quad \texttt{OrGr} \\
 & & & n_{-1} : \quad \mathit{LNode} \\
 & & & \mathit{pCount} : \quad \mathbb{N} \\
 & & & \mathit{firstcall} : \quad \mathit{Boolean} \\
 & & & v_{\mathit{old}} : \quad \mathit{Values} \\
 & & & t_{\mathit{startSession}} : \quad \mathbb{T} \\
 & & & \mathit{sessionLimit} : \quad \mathbb{D}_+ \\
 & & \} &
\end{array}
\tag{5.3}
$$

$$
\begin{array}{lcll}
\mathit{EState} & = & \textbf{struct}\,\{ & \mathit{nodes} : \quad \mathit{ObjSet} \\
 & & & \mathit{values} : \quad \mathit{Values} \\
 & & & \mathit{now} : \quad \mathbb{T} \\
 & & & \mathit{visited} : \quad \mathcal{L}\ S' \\
 & & \} &
\end{array}
\tag{5.4}
$$

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

For this value the following operations are defined:

$$
\begin{array}{l}
\forall\, t \in \mathbb{T} \bullet \\
\quad \infty + t \;=\; \infty \\
\quad \infty - t \;=\; \infty \\
\quad t < \infty \;=\; \texttt{true}
\end{array}
\tag{5.6}
$$

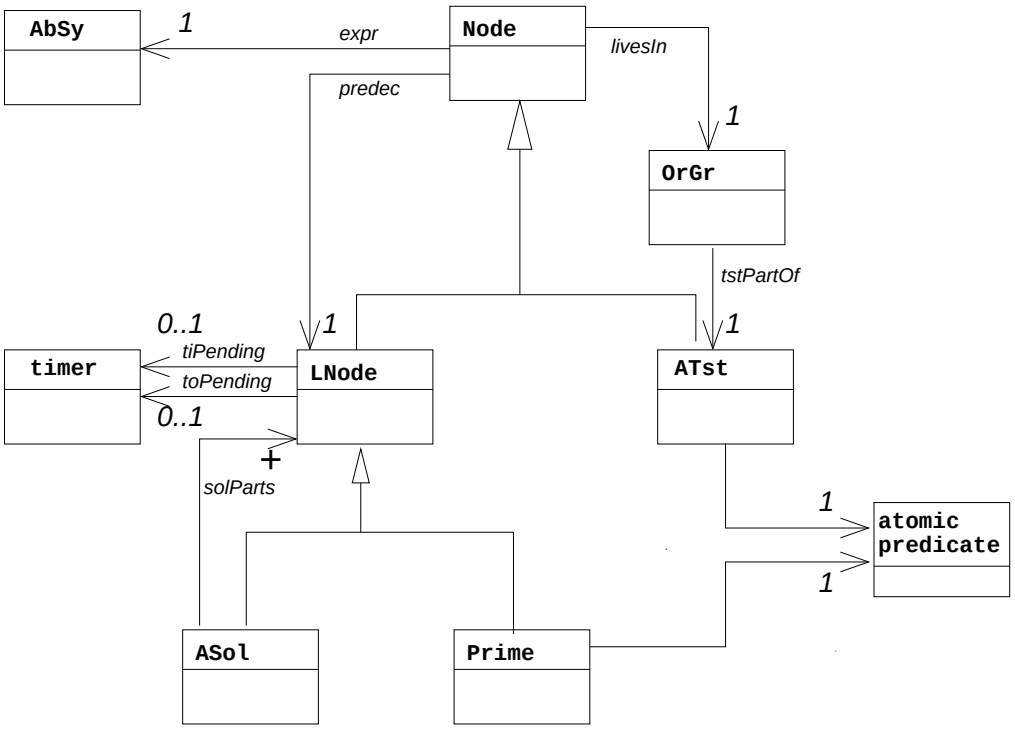
Further combinations are used in the formulæ neither of this nor of the following chapter.

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

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

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

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

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

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

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

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

5.5 Top Level Scheduling Functions

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

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

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

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

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

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

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

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

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

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

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

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

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

5.6 Internal Auxiliary Functions

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

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

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

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

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

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

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

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

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

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

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

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

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

5.8 Creating Solutions of Conjunctions

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

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

$$
\begin{array}{l}
\mathit{installSolution}(s : \mathit{EState}, n : \mathit{LNode}, N : \textbf{set of}\,\mathit{LNode}, A : \texttt{ATst}) : \mathit{EState} \\
\quad = \ \textbf{if}\,\neg\,(a.\mathit{aeFirst} \le a.\mathit{aeLast} \ \wedge\ a.\mathit{minSubSum} \le a.\mathit{maxSubSum})\ \textbf{then}\ s \\
\qquad \textbf{otherwise} \\
\qquad \textbf{if}\ a.\mathit{tiPending} = \infty\ \textbf{then}\ \underline{\mathit{LNode\_becomesValid}}(s', a) \\
\qquad \textbf{otherwise}\ s' \\
\ \textbf{where}\ \ s' = s \oplus (\mathit{nodes} = \mathit{nodes} \cup \{a\}) \\
\qquad a = \texttt{ASol}\,(\ \mathit{solParts} = N \cup \{n\} \\
\qquad\qquad \mathit{aeFirst} = \mathsf{latest}\,(\textbf{map}(\mathit{solParts}, \_.\mathit{seFirst}))\ ) \\
\qquad\qquad \mathit{aeLast} = \mathsf{earliest}\,(\textbf{map}(\mathit{solParts}, \_.\mathit{seLast}))\ ) \\
\qquad\qquad \mathit{minSubSum} = \mathsf{max}\,(\textbf{map}(\mathit{solParts}, \mathit{minSum}(\_))\ ) \\
\qquad\qquad \mathit{maxSubSum} = \mathsf{min}\,(\textbf{map}(\mathit{solParts}, \mathit{maxSum}(\_))\ ) \\
\qquad\qquad \mathit{expr} = \bot \\
\qquad\qquad \mathit{predec} = A.\mathit{predec} \\
\qquad\qquad \mathit{livesIn} = A.\mathit{livesIn} \\
\qquad\qquad \mathit{endReached} = \texttt{false} \\
\qquad\qquad \mathit{sumMinPreds} = A.\mathit{sumMinPreds} \\
\qquad\qquad \mathit{sumMaxPreds} = A.\mathit{sumMaxPreds} \\
\qquad\qquad \mathit{tiPending} = \begin{cases} \mathit{aeFirst} + \mathit{minSubSum} \\ \quad \textbf{if}\ \mathit{aeFirst} + \mathit{minSubSum} > s.\mathit{now} \\ \infty\ \ \textbf{otherwise} \end{cases} \\
\qquad\qquad \mathit{toPending} = \begin{cases} \mathit{aeLast} + \mathit{maxSubSum} \\ \quad \textbf{if}\ \mathit{aeLast} + \mathit{maxSubSum} < \infty \\ \infty\ \ \textbf{otherwise} \end{cases} \\
\qquad\qquad \mathit{seFirst} = \begin{cases} \mathit{aeFirst}\ \ \textbf{if}\ \underline{\mathit{isLeading}}(a) \\ \mathsf{latest}\,\{\mathit{predSeFirst}, \mathit{aeFirst} - \mathit{sumMaxPreds}\} \\ \qquad\qquad \textbf{otherwise} \end{cases} \\
\qquad\qquad \mathit{seLast} = \begin{cases} \mathit{aeLast}\ \ \textbf{if}\ \underline{\mathit{isLeading}}(a) \\ \mathsf{earliest}\,\{\mathit{predSeLast}, \mathit{aeLast} - \mathit{sumMinPreds}\} \\ \qquad\qquad \textbf{otherwise} \end{cases} \\
\qquad\qquad)
\end{array}
$$

$$
\begin{array}{l}
\qquad \mathit{minSum}(n : \mathit{LNode}) : \mathbb{T} \ = \ n.\mathit{sumMinPreds} + \mathit{minDura}(n) \\
\qquad \mathit{maxSum}(n : \mathit{LNode}) : \mathbb{T} \ = \ n.\mathit{sumMaxPreds} + \mathit{maxDura}(n) \\
\qquad \mathit{predSeLast} \ = \ \begin{array}{ll} \mathit{pred}.\mathit{seLast} & \textbf{if}\ A.\mathit{seLastCache} = \infty \\ A.\mathit{seLastCache} & \textbf{otherwise} \end{array} \\
\qquad \mathit{predSeFirst} \ = \ \begin{array}{ll} \mathit{pred}.\mathit{seFirst} & \textbf{if}\ A.\mathit{seFirstCache} = \infty \\ A.\mathit{seFirstCache} & \textbf{otherwise} \end{array}
\end{array}
\tag{5.42}
$$

$$
\begin{array}{l}
\mathit{allCombinations}(N : \mathit{ObjSet}, a : \texttt{ATst}, o : \texttt{OrGr}) : \textbf{set of } \textbf{set of } \mathit{LNode} \\
\quad = \textbf{reduce}\,(\underline{\mathit{tstParts}}\,(N, a) \setminus \{o\}, \mathit{comb}0(N, \_, \_), \{\{\}\}) \\
\mathit{comb}0(N : \mathit{ObjSet}, o : \texttt{OrGr}, \mathit{grown} : \textbf{set of } \textbf{set of } \mathit{LNode}) \\
\quad = \textbf{map}\,(\mathit{finalNodes}(N, o), \mathit{comb}1(\_, \mathit{grown})) \\
\mathit{comb}1(n : \mathit{LNode}, \mathit{grown} : \textbf{set of } \textbf{set of } \mathit{LNode}) \\
\quad = \textbf{map}\,(\mathit{grown}, (\{n\} \cup \_))
\end{array}
\tag{5.43}
$$

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

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

5.9 Termination of Nodes

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

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

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

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

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

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

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

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

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

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

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):

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

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

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

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

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

[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:

$$\begin{array}{ll} \alpha & \\ \wedge & \beta \\ & \hline \beta' \\ & \hline \beta'' \\ & \hline \ldots \\ \gamma & \\ \hline \gamma' & \\ \hline \ldots & \end{array}$$

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.

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

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:

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

[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:

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

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

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

(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 ...

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

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

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

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

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

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

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

... 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:

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

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 ...

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

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 ...

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

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 ...

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

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

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

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 $\{t_0 \dots t_2\}$

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 ...

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

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:

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

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

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

... 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 ...

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

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

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

$$\frac{}{D_{[t_{startSession}\ldots t_{now}]} \in [[(\mathsf{SPath}\ \ n.predec)\ \underline{;}\ n.expHd]]^L}$$

$$\frac{}{D_{[t_{startSession}\ldots t_{now}]} \in [[\mathsf{SPath}\ n]]^L} \qquad (6.34)$$

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 \iff D_{[t_k \ldots t_{now}]} \in [\![^{I_k,A_k} p_k]\!]^L \\
\qquad\quad \wedge\ isValid\ (n_k.predec) \\
\qquad\quad \text{---------- (6.7)} \\
\qquad\quad \forall\ t_{k-1} \mid \mathsf{latest}(n_{k-1}.eFirst, t_k - A_{k-1}) \\
\qquad\qquad\quad \le t_{k-1} \le \mathsf{earliest}(n_{k-1,\boxed{t_k}}.eLast, t_k - I_{k-1}) \\
\qquad\qquad \iff D_{[t_{k-1} \ldots t_k]} \in [\![^{I_{k-1},A_{k-1}} p_{k-1}]\!]^L \\
\qquad\qquad\quad \wedge\ isValid\ (n_{k-1}.predec) \\
\qquad\quad \text{---------- (6.39)} \\
\qquad\quad \forall\ t_{k-1} \mid \mathsf{latest}(n_{k-1}.eFirst, t_k - A_{k-1}) \\
\qquad\qquad\quad \le t_{k-1} \le \mathsf{earliest}(n_{k-1,t_{now}}.eLast, t_k - I_{k-1}) \\
\qquad\qquad \iff D_{[t_{k-1} \ldots t_k]} \in [\![^{I_{k-1},A_{k-1}} p_{k-1}]\!]^L \\
\qquad\qquad\quad \wedge\ isValid\ (n_{k-1}.predec) \\
\qquad \text{---------- } \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\quad \le t_{k-1} \\
\qquad\quad \le \mathsf{earliest}(n_{k-1,t_{now}}.eLast, \mathsf{earliest}(n_{k,t_{now}}.eLast, t_{now} - I_k) - I_{k-1}) \\
\qquad\quad \iff 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\quad\ \wedge\ isValid\ (n_{k-1}.predec) \\
\qquad \text{---------- } \textit{(apply distributivity and associativity of } \mathsf{min}/\mathsf{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\quad \le t_{k-1} \\
\qquad\quad \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\quad \iff 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\quad\ \wedge\ isValid\ (n_{k-1}.predec) \\
\hline
\ldots \\
\textit{(repeat this derivation until } n_0 \textit{ is reached.)} \\
\ldots \\
\hline
\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\qquad n_{1,t_{now}}.eLast - I_0, \\
\qquad\qquad\qquad n_{2,t_{now}}.eLast - (I_0 + I_1), \\
\qquad\qquad\qquad \ldots, \\
\qquad\qquad\qquad n_{k,t_{now}}.eLast - (I_0 + \ldots + I_{k-1}), \\
\qquad\qquad\qquad t_{now} - (I_0 + \ldots + I_k)) \\
\quad \iff 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:

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

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 ...

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

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 ...

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

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

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

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 ...

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

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 $0 \le x \le k$ 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:

$$\frac{\forall\, t_{now} : \mathbb{T}, N : \texttt{ASol}, n : LNode \mid n \in N.solParts \wedge n.expr = {}^{I,A}p_{\_} \bullet isValid\ N_{t_{now}}}{isValid\ n_{t_{now}}}$$

$$\frac{\begin{array}{l}\forall\, t_0 \mid \mathsf{latest}(\ n_k.seFirst, t_{now} - (n_k.sumMaxPreds + A)) \\ \quad \leq t_0 \\ \quad \leq \mathsf{earliest}(\ n_{k,t_{now}}.seLast, t_{now} - (n_k.sumMinPreds + I)) \\ \iff (\ D_{[t_0 \ldots t_{now}]} \in [[\mathsf{SPath}\ n_0 \rightsquigarrow n_k]]^L \\ \quad \wedge\ isValid\ (n_0.predec)\ )\end{array}}{\begin{array}{l}n.sumMaxPreds + A \geq N.maxSubSum \\ n.sumMinPreds + I \leq N.minSubSum\end{array}}$$

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

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

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

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

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)

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

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

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

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

(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)

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

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

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

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

(6.53)

These two results can be combined to ...

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

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 $\{t_0 \dots t_2\}$

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

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

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

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

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:

$$\frac{\begin{array}{l} isValid\,N \\ \hline N.aeFirst + N.minSubSum \;\leq\; t_{now} \\ \hline N.aeFirst \;\leq\; t_{now} - N.minSubSum \end{array}}{\dfrac{(m.sumMinPreds + minDura(m)) \;\leq\; N.minSubSum}{newAEL = t_{now} - (m.sumMinPreds + minDura(m)) \;\geq\; t_{now} - N.minSubSum}} \quad (6.46)$$

$$\frac{\begin{array}{l} N.aeFirst \leq\; t_{now} - N.minSubSum \leq newAEL \\ \hline N.aeFirst \leq\; N_{t_{now}}.aeLast \end{array}}{} \quad (6.61)$$

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 $N.aelast$:

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

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)

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

(2)

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

(3)

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

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:

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

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 $\mathcal{R} \times \mathcal{R} \to \mathcal{R}$.
$\_ \frown \_$	Concatenation of interpretations of type $\mathbf{seq}\,\mathbb{T} \times \mathbf{seq}\,\mathbb{T} \to \mathbf{seq}\,\mathbb{T}$.
$[[e]]^L$	A function from $\mathcal{L}\,S\ \ldots \mathcal{L}\,S''$ into **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

Appendix C

Tutorial On the Writing of Specifications