# Monitoring Temporal Patterns in Guideline-Based Care

eingereicht von
Michael Paesold

DIPLOMARBEIT

zur Erlangung des akademischen Grades
Magister rerum socialium oeconomicarumque
Magister der Sozial- und Wirtschaftswissenschaften
(Mag. rer. soc. oec.)

Fakultät für Informatik, Universität Wien
Fakultät für Informatik, Technische Universität Wien

Studienrichtung Wirtschafsinformatik

Begutachterin
ao. Univ.-Prof. Mag. Dr. Silvia Miksch
Institut für Softwaretechnik, Technische Universität Wien

Wien, im März 2006

# Abstract

Clinical guidelines and protocols have become increasingly important in clinical practice. Computer-based application of guidelines is one of the keys to improved patient care. Therefore, integration of guideline execution into the clinical data flow becomes more and more important. *Temporal data abstraction* is required to apply high-level medical knowledge to low-level measurements and data.

The guideline modelling language Asbru provides strong temporal abstraction capabilities integrated with guideline execution. Because of the complexity of the language, writing an interpreter for Asbru is non-trivial, and an execution engine that would live up to the potentials of the language was not available before.

In this thesis, I describe the design and implementation of a framework to support the execution of Asbru guidelines, building on existing work. Asbru guidelines are compiled into a network of abstraction and plan modules. This network performs the content of the plans synchronised with the patient state.

Monitoring patient data requires algorithms to match the real world data with the temporal patterns defined in the guideline. In order to provide decision support in high-frequency domains such as intensive care, these algorithms must comply with defined runtime constraints. This thesis describes suitable algorithms for the most important temporal abstraction features of Asbru.

The described framework and algorithms were integrated in the Asbru Interpreter and successfully evaluated in a European project.

# Kurzfassung

Medizinische Leitlinien und Protokolle gewinnen in der klinischen Praxis immer mehr an Bedeutung. Die computergestützte Anwendung von Leitlinien ist einer der Erfolgsfaktoren bei der Verbesserung des Gesundheitswesens. Daher wird die Integration der computerunterstützten Ausführung von medizinischen Leitlinien in den klinischen Datenfluss wird immer wichtiger. Temporale Datenabstraktion ist notwendig, um medizinisches Wissen auf konkrete Messungen und Daten anwenden zu können.

Die Planrepräsentationssprache Asbru integriert weit reichende Möglichkeiten zur temporalen Datenabstraktion mit der Ausführung von Leitlinien. Auf Grund der Komplexität der Sprache ist die Entwicklung eines Interpreters für Asbru allerdings nicht trivial. Eine Implementierung, die den Potentialen der Sprache gerecht würde, existierte bisher nicht.

In dieser Diplomarbeit beschreibe ich aufbauend auf bestehenden Arbeiten das Design und die Implementierung eines Systems für die Ausführung von Asbru Plänen. Asbru Leitlinien werden in ein Netzwerk von Abstraktions- und Planmodulen übersetzt. Dieses Netzwerk führt den Inhalt der Pläne in Abhängigkeit von den Patientendaten aus.

Die Überwachung der Patientendaten erfordert Algorithmen um die realen Messungen und Daten mit temporalen Vorgaben in der Leitlinie abzugleichen. Um Entscheidungsunterstützung auch in hochfrequenten Anwendungsdomänen, wie etwa in der Intensivmedizin, verwirklichen zu können, müssen diese Algorithmen strengen Beschränkungen der Laufzeit entsprechen. Diese Arbeit beschreibt solche Algorithmen für die wichtigsten Asrbu-Elemente zur temporalen Abstraktion.

Das beschriebene System und die Algorithmen wurden in den Asbru-Interpreter integriert und erfolgreich in einem europäischen Projekt evaluiert.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Acknowledgements

The first person who I would like to thank is Andreas Seyfang. Many of the ideas and concepts in this thesis originate from the fruitful and often extensive discussions with him. He also was the one who raised my interest in the domain of Artificial Intelligence in Medicine in the first place.

Next, I would like to thank Silvia Miksch for giving me the opportunity to work on this thesis within the Protocure team. I appreciate her support, encouragement, and her trust in me.

Special thanks goes to my wife Julia and my daughter Ksenia for their emotional support and love. I am grateful for the support of my parents Jutta and Dieter Paesold, and my whole family throughout the years.

I thank Peter Votruba from the Protocure team for his contribution to our good work on the Asbru Interpreter. Finally, I thank Andreas Seyfang, Jutta Paesold, and Florian Mitter for their revisions and proofreading of the manuscript.

# Chapter 1

# Introduction

Improving health care has been a significant ambition in history. It is only a logical consequence of the developments of the last century that computers have been integrated into the care process. A great amount of research in *Medical Informatics* (MI) and *Artificial Intelligence* (AI) has been devoted to providing means to facilitate diagnosis, prognosis, and treatment in medicine.

Studies have shown that clinical guidelines and treatment plans have benefits in the practice of medicine [30]. Unfortunately, clinicians are often not familiar with written guidelines and do not apply them appropriately during the actual care process [84]. Nevertheless clinical guidelines can improve patient care if properly developed, communicated, and implemented [37].

To support the computer-based application of guidelines, many frameworks for design, validation, and execution of clinical guidelines have been developed. The Arden Syntax [33] was the first language for the representation of clinical knowledge in decision-support systems. The limitations of Arden and other early approaches have resulted in the development of a wide range of modeling methodologies available today.

Diagnosis and treatment have usually been seen as distinct tasks in computer-based decision support, but if both are tightly integrated, each task can support the other one [61]. The guideline modeling language Asbru [60] provides the expressive power to represent the abstract temporal concepts required for both tasks. The foundation of these concepts is knowledge-based temporal data abstraction and skeletal plan execution.

In this thesis, I describe the design and implementation of a framework and several algorithms to support temporal data abstraction in the context of guideline-based care. My work is based on the framework proposed by Seyfang [59]. The implementation is the basis for the Asbru Interpreter developed as part of the Protocure II project[1], which aims at integrating formal methods in the life cycle of guidelines.

[1] See the project homepage at `http://www.protocure.org`. Accessed Feb 2, 2006.

## 1.1 Clinical Guidelines and Protocols

"Clinical guidelines are systematically developed statements to assist practitioner and patient decisions about appropriate health care for specific clinical circumstances" [22]. Clinical guidelines usually address a specific health condition and provide recommendations to the physician about investigation, diagnosis and treatment. The difference between "guideline" and "protocol" is not well-defined, but a clinical protocol is basically a more detailed, site-specific version of a guideline, referring to a certain class of therapeutic interventions [41]. In the context of this work I use the term guideline most of the time, but the statements are true for guidelines as well as protocols if not explicitly stated otherwise.

## 1.2 Objectives

The Asbru language provides strong temporal abstraction capabilities integrated with guideline execution. Unfortunately, an execution engine that would live up to the potentials of Asbru was not available before. None of the previous implementations closely integrate plan execution and the required data abstraction into the clinical data flow in a high frequency domain such as intensive care.

To fill this gap, a new execution engine for Asbru, the *Asbru Interpreter*, is being developed. The theoretical work of Seyfang provides the foundation for this implementation [59] and my work. There are two main objectives for this thesis. The first is to analyse, design, and implement a framework for a modular execution engine for Asbru, including an exact definition of the semantics of the components and the interactions between them. Monitoring patient data requires algorithms to match the real world data with the temporal patterns defined in the Asbru plan library. In high-frequency domains such as intensive care, these algorithms must comply with defined runtime constraints. This requires new algorithms. The second main objective is to design and implement algorithms for monitoring temporal patterns using the created framework.

## 1.3 Outline of this Thesis

After having presented the motivation and background for this thesis, I give an outline of this thesis here.

**Chapter 2** gives an overview of the related work on computer-interpretable clinical guidelines as well as the temporal aspects of modeling medical knowledge.

**Chapter 3** introduces the reader to the main concepts of Asbru with a focus on the temporal aspects.

**Chapter 4** gives an overview of the Asbru Interpreter from a software design point of view. It introduces the basic concepts of the module framework and its implementation in an object oriented language. An example from the field of artificial ventilation of neonates provides a link from the conceptual level to execution.

**Chapter 5** describes algorithms developed for monitoring temporal patterns in patient data. Algorithms for Asbru's most important temporal abstraction features are presented.

**Chapter 6** concludes the thesis with an evaluation of the implementation, a summary, and an outlook on possible future improvements.

# Chapter 2

# Related Work

This chapter gives an overview of the related work on computer-interpretable clinical guidelines and protocols as well as on the temporal aspects of modeling medical knowledge. The chapter is divided into two main parts. The first part describes several guideline modeling methodologies, the second part focuses on temporal data abstraction and intelligent data analysis.

## 2.1 Guideline Modeling Methodologies

To execute clinical guidelines in a computer-supported way, free-text guidelines complemented with tables, formulas, or flow charts, have to be formalised and translated into an executable form. Many researchers have proposed frameworks for design, validation, and execution of clinical guidelines.

This section discusses different guideline modeling methodologies that support computer-based execution. Besides a general review of each approach, I focus on the analysis of the temporal features provided by different approaches. A general in-depth comparison of several methodologies, including an analysis of the structures used to model guidelines, can be found e. g., in [54].

### 2.1.1 Asbru

Asbru is a time-oriented plan representation language that represents clinical guidelines as skeletal plans [60]. Skeletal plans are plan schemata at various levels of detail, capturing the essence of the procedure, but leaving room for execution-time flexibility in the achievement of particular goals [25]. In Asbru, the concept of skeletal plans has been enriched with temporal aspects. Several knowledge roles are attached to a plan: preferences, intentions, conditions, effects and a plan body, which describes the actions to be taken.

Asbru was originally developed in the Asgaard project [65, 45]. Since then, Asbru has evolved into an XML-based language with many conceptual

extensions [60]. A partial definition of the semantics of Asbru has been created using Structured Operational Semantics. An overview is available in [5].

Asbru's distinguishing features are:

- Intentions, conditions, effects and world states are temporal patterns, which allow reasoning about the contained knowledge.
- Actions and states can be continuous (durative).
- The language allows to model temporal uncertainties, different granularities, and repeated patterns in events, actions, plans and world states.
- Plan can be executed in a fixed sequence, in parallel, or unordered (with and without mutual exclusion).
- Because of the advanced (temporal) data abstraction capabilities, diagnosis and treatment can be tightly integrated allowing each one to support the other [63].

All conditions for the transition from one plan state to another are expressed in terms of temporal patterns. A temporal pattern consists of one or more parameter propositions or plan-state descriptions. Each parameter proposition contains a value description, a context, and a time annotation. The time annotation used allows a representation of uncertainty in starting time, ending time, and duration of an interval. Start and end are defined as shifts from a reference point. Reference points can be defined as sets of cyclical time points or references to parameter changes, allowing for repeated temporal patterns.

Since Asbru plans cannot be easily modeled using the classical approach of flow charts, new visualisation paradigms are needed to represent Asbru plans. Several tools such as AsbruView [39] and CareVis [1] have been developed for the visualisation task. Other tools are available to translate informal, textual guidelines to an intermediate representation [64] and to the fully formalised Asbru XML representation [85].

There have been two attempts to implement an execution engine for Asbru. The first implementation created by Bosse [9] translated the guideline into a representation suitable to be executed in Clips. The implementation was customised for a single clinical protocol and was therefore abandoned after the end of the project. A more general implementation is AsbruRTM [27,26]. It has been used to test Asbru guidelines in intensive care. Unfortunately, AsbruRTM only supports a subset of the available plan types in Asbru Light (again only a subset of the full Asbru language) and does not integrate advanced temporal data abstraction.

Spock is a system for application of guidelines in Hybrid-Asbru, which is a semi-formal guideline language that combines formal structure with description text [87]. Spock is therefore not suited for fully-automated execution – it can support a human agent applying a guideline. Spock is integrated with the IDAN architecture [8] and can utilise its temporal abstraction capabilities.

Asbru is the representation language used in this thesis, therefore, a more detailed introduction to its features can be found in chapter 3.

### 2.1.2 EON

EON was developed at Stanford University between 1996 and 2003 as a component-based, extensible architecture, and provides a suite of models and software components for creating guideline-based applications [46]. EON uses a task-based approach to define decision-support services that can be implemented using alternative techniques. Four tasks can be identified: knowledge acquisition, protocol-based decision support, temporal abstraction, and explanation services.

Knowledge acquisition is done using Protégé representing knowledge in three models. The generic Dharma guideline model is used to create and maintain clinical guidelines and protocols. The domain ontology is represented in the medical speciality model. A patient data model represents the patient information.

Guideline execution is implemented in the PADDA guideline server [80]. Clinical professionals interact with the PADDA server through the PADDA client. The temporal-abstraction task is solved using the Knowledge-Based Temporal-Abstraction method [66] implemented in RÉSUMÉ (see section 2.2.4). Originally RÉSUMÉ and Chronus, a temporal query system, were directly implemented as parts of the guideline execution component. Later, the Tzolkin [47] temporal mediator (see section 2.2.4) was integrated for the temporal abstraction task to provide temporal query and abstraction services [81]. The temporal abstraction component interfaces with a relational database management system to access patient data as series of time-stamped values.

Explanation services, which can be used by other components, are implemented in the WOZ [67] explanation server.

EON provides three models to specify decision criteria. Simple expressions such as Boolean operators can be defined using simple templates. Advanced decision criteria are offered through the Protégé Axiom Language (PAL) or through temporal queries as provided by the Tzolkin component.

The EON architecture is used in the ATHENA Decision Support System (DSS). ATHENA is a guideline-based decision support system for hypertension management [28,29].

Although EON provides advanced temporal abstraction capabilities, it

seems that it cannot be used for planning in high-frequency domains, because of the database-centric approach and the performance limitations of the temporal abstraction component. Although the component-based architecture would allow an extension for such a usage scenario, to my knowledge there haven't been efforts in that direction.

### 2.1.3 GLARE

GLARE is a domain-independent system with tools for the full life-cycle of clinical guidelines [76]. The name stands for GuideLine Acquisition, Representation, and Execution. It has been developed at the Universitá del Piemonte Orientale "Amedeo Avogadro", Alessandria, Italy.

The representation formalism builds on a limited but comprehensible set of primitives and is designed to cope with different types of temporal constraints. Particular attention was paid to the role of periodic events. The following temporal concepts are considered [75] and solved as instances of *Simple Temporal Problems* (STP) [18]:

- Specification of temporal bounds on differences (e. g., the time between two successive intakes of a specific medicine should not exceed eight hours during the first two days of treatment).
- Temporal constraints between actions belonging to the same plan.
- Temporal constraints in different plans (e. g., checking of vital signs must be repeated during the whole surgery).
- Temporal constraints on repeated actions (e. g., a specific medicine has to be taken every eight hours for one week).

Temporal reasoning about these constraints supports guideline creation as well as execution. When a guideline is being acquired, reasoning can be useful for instance to check the consistency of the constraints imposed on actions. At execution time, temporally minimal procedures can be determined.

GLARE provides a representation formalism, a knowledge authoring tool, and a guideline execution tool. The formalism is designed to be easily understandable so that guidelines can be designed by trained domain-experts and there is no strong requirement for knowledge engineers. The authoring tool provides different types of checks to help developing consistent guidelines.

During guideline execution GLARE provides another feature for decision support: the "what if" facility, which allows hypothetical reasoning [77]. Using this facility, it is possible to compare different paths in the guideline, by simulating what could happen if a certain choice was made. Users are thereby helped in gathering various types of information needed to discriminate alternatives.

Guidelines are currently directly stored into a relational database. It is therefore difficult to evaluate the specific features of the guideline representation. According to [78], an XML representation is worked on. This should ease the evaluation of GLARE by other groups.

Although GLARE supports temporal constraints on and temporal reasoning about actions and plans, publications on GLARE do not talk about temporal abstraction capabilities for diagnosis and prognosis.

### 2.1.4 GLIF

The Guideline Interchange Format (GLIF) stresses the importance of sharing guidelines among different institutions and systems [52, 10]. As the name suggests, the original idea for GLIF was a format that other guideline representations could be translated into. Because this aim seemed infeasible, the focus was shifted to build a standard format for sharing guidelines. GLIF has been collaboratively developed by groups at Columbia, Stanford and Harvard Universities (the InterMed Collaboratory).

GLIF version 2 enabled modeling a guideline as a flowchart of structured steps, representing clinical actions and decisions. However the attributes of the constructs were defined as free text so that such guidelines could not be used for computer-based execution. GLIF3 builds upon GLIF2 but remedies its main deficiencies [52].

Guidelines in GLIF3 can be designed at three levels:

- A conceptual, flow-chart based representation that is human-readable.
- A computable representation that allows a computer to interpret the logic and sequence specified in guidelines. Guidelines at this level can be verified for logical consistency and completeness.
- An implementable specification for incorporation into clinical systems. This level allows adaptation to the existing information systems in different institution by non-shareable elements.

GLIF3 is an object oriented language. The model is described using Unified Modeling Language (UML) class diagrams[1]. The guidelines are stored in an RDF-based[2] XML representation.

Decision criteria can be written in an expression and query language. The latest development, GELLO [50], is an object oriented version of GEL [53], the "guideline expression language", which itself is a superset of the Arden Syntax logic grammar. The default medical data model of GLIF is based on the HL7 Reference Information Model (RIM). GELLO can access all information in the "virtual medical record" compatible with HL7 RIM.

[1] UML is defined by the Object Management Group, `http://www.uml.org/`, accessed Feb 17, 2006.

[2] Resource Description Framework, `http://www.w3.org/RDF/`, accessed Feb 17, 2006.

GLIF provides a layered model for representing medical knowledge. The core GLIF layer provides a standard interface to all medical data and concepts that may be represented and referenced by GLIF. The RIM layer provides a semantic hierarchy for medical concepts, and allows attribute specification for each class of medical data.

The execution engine named GLEE can be used to execute GLIF3 guidelines. Neither the layered model nor the GELLO language are currently supported as the software is not yet to be integrated in clinical information systems [86].

### 2.1.5 GUIDE

Guide, a component-based multi-level architecture, integrates the formalised model of medical knowledge contained in clinical guidelines with workflow management, formally grounded on Petri nets [56]. It has been developed at the University of Pavia, Italy.

The original system [56,57] was comprised of three components: an editing tool to support clinical experts in specifying guidelines in a formal representation, a translation tool from the guideline into a high-level workflow model, and a low-level workflow builder to model the medical procedures according to site specific requirements.

Similar to other methods, the model allows the decomposition of tasks in subtasks to describe different levels of abstraction. The medical guideline is seen as the high-level model, the organisational work flow as the low-level model. Both are represented and implemented as an integrated system using workflow management methods and tools. The use of time constraints allows the modeler to represent how long it may take for a task to be executed.

The workflow model is translated into a Petri net for simulation purposes. The simulation allows to validate the careflow model and to analyse the resource allocation and time constraints in the workflow. For real-world deployment, the system is implemented in a commercial workflow management system and linked with electronic patient record systems.

The original Guide approach has been superseded by the *NewGuide* project [13,14]. With the focus still on careflow, guideline management and careflow management have been decoupled and are now only loosely linked through message-based communication[3].

The system architecture defines the careflow management system and the guideline management system. Both system parts use the shared organisational and medical ontologies. The guideline management system is comprised of a guideline editor, a central guideline repository, the inference engine, and a reporting system. The use of the SOAP message framework

[3] The message system is implemented using SOAP (see `http://www.w3.org/TR/soap/`, accessed Feb 11, 2006).

and the definition of a virtual medical record eases the integration with legacy systems.

The focus of the project on careflow management allows to research specific aspects of applying guidelines to the clinical environment, such as compliance or non-compliance to guidelines. The decoupling of "pure medical action" and "organisation actions", i. e. those actions that could be managed by a workflow management system, allows to handle exceptions that can arise during the guideline-based process [13].

Regarding the temporal aspects of Guide, the representation formalism allows the specifications of rules and criteria in an object-oriented language that can also manage qualitative and temporal abstractions [13]. Temporal abstractions are based on the work by Belazzi et al. [40], which seems to imply that temporal abstraction capabilities are again an external resource.

### 2.1.6 PRODIGY

PRODIGY is a decision support system that integrates with primary care information systems [55,36]. Its main focus is to support general practitioners in prescribing. PRODIGY was introduced in 1996 at the University of Newcastle upon Tyne and is under continued development. The system is used by a large number of general practitioners in the United Kingdom, in form of clinical information systems developed by two different vendors [35]. Phase three of the project added support for chronic disease management.

PRODIGY models guidelines as a series of decisions that a general practitioner may have to make in different patient encounters. The model is based on patient scenarios, which represent entry points into a guideline and facilitate a patient's automatic entry into an appropriate plan or subplan. The basic idea of patient scenarios was adopted by EON and GLIF. Protégé is used as knowledge acquisition tool.

Criteria, as in conditions and preferences, are implemented as Boolean expressions to express a preference for or against a choice. A criteron may use quantitative and qualitative expressions, it may contain a reference to a time period, e. g., "absence of cough within last 3 months". The criteria are matched with the data in the electronic patient record (EPR). A set of functions is provided with which abstract concepts, such as the risk rating, can be computed from more elementary concepts such as smoking, cholesterol level, and other risk factors [36].

PRODIGY does not aim at fully-automated guideline execution and always requires confirmation of the physician.

### 2.1.7 PRO*forma*

PRO*forma* is a knowledge composition and process modeling language supported by acquisition and execution tools, with the goal of supporting guide-

line dissemination in the form of expert systems that assist patient care through active decision support and workflow management [24]. It has been developed at the Advanced Computation Laboratory of Cancer Research, UK since 1992. PRO*forma* combines logic programming and object-oriented modeling, formally grounded in the $R_2L$ language.

One aim of the PRO*forma* project is to explore the expressiveness of a deliberately minimal set of modeling constructs. This is also shown in the fact that during the onward development of PRO*forma*, the number of possible plan states has been *reduced* to the minimum set necessary to provide the required behaviour (from eleven to four states). The other "states" have been determined to be inferable from other properties of a task.

The project also extensively considered the topic of safety in systems development and the risks of agent technologies, in which expert systems are built that act autonomously. This includes the development of a knowledge representation language (RED). RED and its implementation $R_2L$ are hybrid rule- and knowledge-based languages that explicitly represent evidence for assertions and use classic logic to form rules [23].

A semantic specification of PRO*forma* exists that provides operational semantics for the language through the description of an abstract PRO*forma* engine along with a set of "public operations" that may be performed on the engine by an external system [74]. Rules are set out that describe how the public operations change the state of the abstract engine. Through the semantic specification the authors hope to encourage tools and software development using the PRO*forma* language by other groups as well.

A number of components have been written to create, visualise, and enact PRO*forma* guidelines. The *Tallis* software suite includes components written in Java. PRO*forma* has also been used as the basis of a commercial decision support and guideline technology *Arezzo* from Infermed Ltd., London, UK.

Similar to Asbru, effects of plans are implemented in a computer interpretable form, although PRO*forma* lacks the expressive powers to describe temporal uncertainty of effects.

## 2.2 Temporal Reasoning in Medicine

Time is an important aspect of the real world. Events occur at some point in time, facts are true over time, objects exist over time. Relationships between events, facts, and objects exist over time [51]. Time is so fundamental that reasoning about time often occurs unnoticed. The same can be observed when considering decision making in medicine [4].

When modeling information or processes in computer systems, we must be able to model temporal information and change over time. Representation of and reasoning about time has been an active direction of research for the last decades. Section 2.2.1 introduces the most important aspects of

representation of time.

Two areas of research identified in [16] are especially important for our work: *temporal data maintenance* (traditionally linked with the (temporal) database community and described in section 2.2.2) and *temporal reasoning* (traditionally linked with the AI community, see section 2.2.3). The focus of temporal data maintenance is the storage and retrieval of data in a way that allows the representation of the time-oriented aspects of the data. Temporal reasoning addresses reasoning about the state of the world in a time-oriented way.

Retrieval of time-oriented data as well as temporal reasoning utilise techniques summarised as *temporal abstraction*. Temporal abstraction, or more exactly temporal *data* abstraction, deals with the context-dependent, time-oriented transformation of raw data into higher-level concepts.

Intelligent Data Analysis (IDA) refers to all methods that are devoted to support the transformation of data into information exploiting the knowledge available on the domain [7]. It usually includes, but is not limited to, temporal abstraction. Intelligent Data Analysis is a fundamental aspect of clinical decision support. Therefore, most of the related work represented here can also be classified as IDA.

An area of research where temporal reasoning plays an important part is guideline-based care (compare section 2.1).

### 2.2.1 Representation of Time

There are several aspects that are considered in the discussion about representing time:

- the basic "shape" of temporal structure,
- the "topological" perspective,
- the basic reference: constant-based or interval-based view of time, and
- modeling of change.

Usually time is conceived as a line on which temporal references can be aligned. In that case, time is linear and the set of time points is completely ordered (e. g., in [3]). Another popular approach is a future-branching structure that allows to represent hypothetical alternatives. A past-branching structure could provide a framework for abductive reasoning. Circular time can be useful to describe repetitive patterns or cyclic processes [4].

When time is considered from a topological perspective, i. e. as discrete, dense, or continuous, it can be analysed under the light of topology in mathematics [4]. Time references can be defined as discrete succession, i. e. isomorphic to $\mathbb{N}$ or $\mathbb{Z}$. Other problems may require a continuous conception of time, i. e. isomorphic to $\mathbb{R}$.

A much discussed aspect of time is the basic temporal reference to use. Time can be represented using an instant-based approach that uses time-points to refer to punctual occurrences [68]. It can also be represented period-based using intervals as the basic reference [2,3]. Some approaches combine these two [83].

It is important to note that the question is not only one of representation. Intervals can be represented as start and end points, in the same way, time points can be seen as infinitely short intervals. The distinction is really one of semantics, as recently discussed in the context of temporal databases [79]. This understanding is common knowledge in linguistics and has been applied to Artificial Intelligence before (e. g., [66]). In linguistics, sentences are classified within different *"aktionsart classes"* (e. g., *activities, accomplishments, achievements, and states*). Whereas for a *stative* sentence it is true that the statement is true for any subinterval (e. g., if a person is asleep between 1:00 and 8:00 a.m., she is also asleep between 3:00 and 4:00 a.m.), the same is not true for an *accomplishment/achievement* (e. g., if someone built a house in the interval from September 1 until June 1, then it is false that she *built a house* in any subinterval of this interval). Terenziani et al. show that a point-based semantic is unable to describe such *accomplishments/achievements*. They describe a model for temporal databases that handles both kinds of semantics [79].

When talking about change, there are two main ways to make temporal references. One can use either absolute time references like "on February 28, 2006" or relative references like "as soon as possible" or "after each meal". In the first case, time is conceived to have existence from the very beginning and the concept of state and change are derived from it. The other concept relies on the notion of change itself, which is directly associated to events. Change is considered as the fundamental concept and time must be built from it [4]. In the area of temporal databases, both approaches have been used (see for example *TSQL2* [70] and the *Event calculus* [83]).

### 2.2.2 Temporal Data Maintenance

Temporal data maintenance, the time-oriented storage and retrieval of data, is usually linked to *temporal databases*. There have been several developments in the medical informatics community as well as general approaches in the (temporal) database community, which will be described in the following.

Of general importance is the work of Snodgrass et al. [71], who identified three different temporal dimensions:

**Transaction time** (or physical time) is the time at which data is stored into a database or entered into a system.

**Valid time** (or logical time) is the time at which the data is valid in the

real world, for example the time when a blood sample is taken.

**User-defined time** is any other time dimension that is application-dependent and has therefore no special meaning to a temporal database system, for example the time when the blood sample is analysed in the laboratory.

Snodgrass also defines four types of databases distinguished in their support for these temporal dimensions. *Snapshot* or *static databases* are based on a flat, timeless model, they only have a notion of the current state. *Rollback databases* keep the history of the recorded data but consider only the transaction time. *Historical databases* are based on a model of the valid time, they can present the past as it is seen at the present. Finally, *bitemporal databases* (originally just *temporal databases*) are a combination of historical and rollback databases and explicitly model both transaction time and valid time. This allows to query the current view of history as well as the view of history from any past time point.

The *bitemporal* model was the basis of *TQuel* (Temporal Query Language) [69], and latter *TSQL* (Temporal Structured Query Language) as well as *TSQL2* [70]. *TSQL2* was integrated into the *SQL-3* standard as *SQL/Temporal* [72].

Temporal data maintenance approaches related to the medical informatics community are, e. g., GCH-OSQL/GCH-OODM, or Chronus. The Granular Clinical History-Object Structured Query Language (GCH-OSQL) is an object-oriented temporal query language for temporal databases. GCH-OODM is the corresponding object-oriented temporal data model. The language is designed to support mixed temporal granularities and integrates temporal abstraction capabilities [15].

Chronus is a query system supporting temporal extensions to SQL for relational databases. It provides an extension not only to the query language, but to the relational algebra for managing temporal information. It is built as a separate component to provide temporal maintenance capabilities on top of a *historical* relational DBMS. Its query language is called *Time Line SQL* (TL-SQL) [17]. Chronus has been integrated with RÉSUMÉ in the Tzolkin system (see section 2.2.4).

### 2.2.3 Temporal Reasoning

Temporal reasoning has been used in medical domains in a number of different tasks, such as diagnosis, prognosis, interpreting, critiquing, and therapy planning. These tasks are often interdependent and share basic techniques such as *temporal abstraction*.

From a methodological point of view, temporal reasoning in the medical domain can be classified as deterministic or probabilistic approaches. Deterministic approaches are based on either well-known formalisms from AI,

ad-hoc rules, or ontologies. Probabilistic approaches are typically associated with the interpreting or prognosis tasks under uncertainty (including e. g., Bayesian belief-networks) [16].

A comprehensive proposal for temporal reasoning is the Knowledge-Based Temporal-Abstraction (KBTA) method [66]. It defines several domain independent subtasks of temporal abstraction:

1. *Temporal context restriction* creates relevant interpretation contexts required for focusing and limiting the scope of inference. Knowledge must only be applied in the specific context where the knowledge is correct.

2. *Vertical temporal inference* obtains higher-level concepts based on contemporaneous data, e. g., inferring a general state description from individual parameters.

3. *Horizontal temporal inference* determines the domain value of an abstraction created from different joined abstractions (attached to different time intervals).

4. *Temporal interpolation* bridges gaps between similar-type disjoint episodes to create longer intervals using domain-specific knowledge.

5. *Temporal pattern matching* is matching of predefined complex temporal patterns with the patterns created by the abstractions.

Problem-solving methods for these tasks have been implemented in several systems, e. g. the RÉSUMÉ system. A particular feature of the KBTA method is that the temporal abstraction methods are independent of the domain-specific knowledge. The relevant domain knowledge is modeled in the domain ontology. Therefore, it is possible to reuse the KBTA method in many domains. Additionally, the method does not rely on predefined patterns, but instead automatically creates abstractions according to the domain ontology.

The KBTA method is not the only approach to temporal reasoning. The next sections describe the research on and applications of temporal reasoning and Intelligent Data Analysis.

### 2.2.4 Applications of the Knowledge-Based Temporal-Abstraction Method

This section describes applications of the Knowledge-Based Temporal-Abstraction (KBTA) method [66]. There have been several different approaches to benefit from this general framework for temporal reasoning.

### RÉSUMÉ

RÉSUMÉ is the first implementation of the KBTA method. It is implemented in CLIPS and is composed of a temporal-reasoning module, a static domain knowledge base (the domain's temporal abstraction ontology), and a dynamic temporal fact base for input and output data.

Abstractions are derived from input data, these abstractions and the input data are again used to derive or retract new abstractions in a recursive process. A *truth-maintenance* system is implemented so that changes in abstractions are propagated correctly by maintaining logical dependencies among parameters and derived abstractions. This is required to process data out of temporal order (e. g., analysis of a blood sample in a clinical laboratory might by available only several days after the blood sample was taken). The dynamic temporal fact base is therefore essentially a historic database – it allows to change the present view about the past, based on new facts.

The domain ontology is represented in the parameter-properties ontology, which describes the properties of parameters in the domain, and the event and context ontologies. Classifications and functional knowledge are represented as either tables or "black box" functions. All domain-specific knowledge is therefore modeled in the domain knowledge base. RÉSUMÉ itself is domain-independent.

RÉSUMÉ has been tested in several domains including the domain of therapy for insulin-dependent diabetes. It has also been built into the Tzolkin system. CAPSUL [11] extends RÉSUMÉ with a language for the specification of periodic patterns. The main problem of RÉSUMÉ is its performance, as runtime and space complexity are exponential to the data set size [48].

### The Tzolkin System

The Tzolkin system integrates temporal reasoning with temporal data maintenance [47]. The approach is called a *temporal database mediator*. However, the name *temporal abstraction mediator* would be more appropriate, because the software acts as a mediator between clinical applications and a database management system providing *temporal abstraction* capabilities. Tzolkin consists of a temporal reasoning component (RÉSUMÉ), a temporal maintenance component (Chronus), a query preprocessor, and a controller.

Tzolkin allows to query not only the data already available in the existing clinical DBMS, but also dynamically generated time-oriented abstractions of the data using the abstraction capabilities of RÉSUMÉ. Its query language, *SQL for Abstractions* (SQLA), is a syntactic as well as semantic superset of TL-SQL, the query language of Chronus. Additionally to the GRAIN and WHEN query extensions of TL-SQL, SQLA offers a CONTEXT extension that defines the abstraction context for RÉSUMÉ.

Tzolkin was used as the temporal abstraction component of the EON decision-support architecture (see section 2.1.2). However, its main problem is performance. The combination of both database and rule-based methods leads to scalability problems in this approach. Basically, RÉSUMÉ generates all abstractions mentioned in the query and writes them into a temporary database. Then Chronus uses this temporal database to execute the query.

A *batch computation* mode was implemented to counter the performance problem. Batch computation generates and stores all possible abstractions for a given set of data in advance. The inherent problem with this method is that the truth-maintenance of RÉSUMÉ does not cover the external database. Therefore, the precomputed abstractions are in fact only a snapshot of the data and subsequent changes in the underlying clinical database will lead to the combination of valid and invalid data in the system. The authors conclude that without a truth-maintenance system at the database level, caching features are infeasible.

### RASTA

RASTA is a distributed temporal abstraction system to facilitate knowledge-driven monitoring of clinical databases [48]. RASTA is a recursive acronym for **R**ASTA: **A** **S**ystem for **T**emporal **A**bstraction. The main motivation for the development of RASTA was to have a replacement for Tzolkin (and therefore RÉSUMÉ) as the temporal reasoning component in EON. In the words of the authors, "RÉSUMÉ does not offer real-time response rates for anything other than small single-patient data sets. Its fundamental problem is that there is an exponential relationship between the size of the data set it operates on and its memory and CPU requirements." [48]

RASTA implements a part of the KBTA method using a distributed algorithm. Individual abstractions are evaluated in an abstraction hierarchy, the abstraction steps can be distributed over several computers. The algorithm is data-driven, so it does not require complicated synchronisation between the abstraction processes. From a functionality point of view, RASTA is similar to RÉSUMÉ. It does not implement ad-hoc temporal queries like Tzolkin does. RASTA is implemented in Java.

The problem of *truth maintenance* is implemented using database triggers informing RASTA when data has been modified. In that case, the corresponding changes in the abstractions are propagated through the hierarchy. This process may involve retracting abstractions, which can be computationally complex.

When distributed over many computers, RASTA supports very large data sets. It has been used as the temporal reasoning component for EON in the ATHENA decision support system [28].

### Chronus II

Chronus II is a *temporal database mediator* that provides temporal query capabilities on top of regular (snapshot) database management systems [49]. It uses a historical database model and is based on Chronus and parts of the TSQL2 temporal query specification. Additionally, it integrates RASTA and thereby provides temporal abstraction capabilities.

The Chronus II design is similar to that of the Tzolkin system. First, RASTA is used to create abstractions of the raw-data, then Chronus II can apply the query to the created abstractions. Like RASTA, Chronus II is implemented in Java. It can be integrated with any standard DBMS that provides a JDBC connector.

Chronus II has been used in the ATHENA decision support system for temporal queries. The language has also been used as temporal predicate language in a guideline modeling system [49].

### Momentum

Momentum is an active time-oriented database for temporal abstraction, exploration and analysis of time-oriented data. It is an answer to the performance problems inherent to the former approaches, namely Tzolkin and Chronus II [73]. The inefficiency of Tzolkin has been already discussed above. Chronus II has similar problems as it works basically the same way. The difference is only that RASTA can be used in real-world scenarios, because it can be used on a distributed architecture.

Momentum replaces the temporal database mediation approach with an integration of temporal reasoning and temporal data maintenance in one system. It is implemented as a time-oriented Active Database (ADB). When primitive data is added to the system, defined rules incrementally create or retract abstractions. This is similar to the *batch computation* mode of Tzolkin, but is performed dynamically and incrementally. Initially, data is loaded in an extract, transform, and load (ETL) process from outside data sources. Updates are done in a delta-based update ETL process. These concepts are borrowed from data warehousing, and in fact, the architecture of Momentum can be seen as a special form of a data warehouse.

In addition to the *temporal abstraction query and instruction language* (TAQILA), a knowledge definition language (KDL) to define domain specific knowledge according to the KBTA ontology has been created. The TAQILA language supports both entering raw-data into the system as well as querying raw-data and abstraction concepts. It allows the explanation of derived concepts and dynamic sensitivity analysis using the truth-maintenance system ("what if?"). The TAQILA language is not SQL-based.

Momentum is implemented in Java. KDL, TAQILA, and query results are represented in XML. The data is stored into an XML:DB compliant

native XML database. It is planned to further improve the performance by making the abstraction algorithms parallel and distributed. The system has not yet undergone clinical evaluation, according to the latest publications.

#### The IDAN Architecture

IDAN is a framework for distributed mediation of temporal-abstraction queries to clinical databases [8]. It is task-specific and domain-independent. IDAN consists of time-oriented data sources, domain-specific temporal-abstraction knowledge sources, temporal abstraction services, and a mediator to integrate all services. The default abstraction service is ALMA. It is based on a temporal-deductive database and logical programming (implemented in a Prolog interpreter).

The easiest way to describe how IDAN works, is to describe the (simplified) query processing flow. An application sends a query to the mediator. The mediator retrieves relevant concepts from the knowledge sources required for the query. Then it requests the raw data from the data sources, converts the returned data, and passes the data and the knowledge representation to the abstraction services (ALMA by default). ALMA forms temporal-abstraction rules from the knowledge, processes the query, and returns abstract-concept answers. These answers are handed back to the application.

IDAN integrates several standards for medical vocabulary for diagnosis and procedures. It is implemented as a number of web services using .NET technology. Queries, data, and results are represented in XML. The mediator itself is implemented in the C# programming language.

The focus of IDAN is to provide web services that can be used in many different applications. The architecture is quite different from the Momentum architecture. Although IDAN is a more generic solution, it is less capable of the task of continuous monitoring or answering queries about large patient populations.

### 2.2.5 Other Applications of Temporal Abstraction

A lot of the research presented in this subsection is influenced by the KBTA *theory*, but since none of these proposals implement the KBTA *method* in the same way as proposed in [66], they are presented in their own section.

Similar to the IDAN architecture (but much earlier), Larizza et al. [40] use an HTTP based server for temporal abstractions. It is similar to IDAN in the sense that it is implemented as a web service (although the term *web service* was not yet introduced at that time). The domain-independent implementation provides *basic* temporal abstractions such as trend abstractions, state abstractions and *complex* abstractions, for example temporal operators defined in the Allen interval algebra. It handles data of different granulari-

ties. The scalability of the system must be questioned as all input-data must be sent to the server for each individual abstraction request. The same set of temporal abstractions has been used by Bellazzi et al. for the assessment of clinical performance of a hemodialysis services [6].

Another important area of research is decision support for intensive care units (ICU). In this area, there is an intensive use of equipment and sensors. Therefore, the problem is not only one of information and knowledge extraction, but also of varying quality of sensor data and efficient design and implementation. The data is usually hard to interpret even for expert physicians.

One aspect that is common in research in this area is trend detection. $TrenD_x$ [31] is a trend diagnosis system that provides assistance to medical diagnosis. It matches patient data to patterns of normal and abnormal trends called *trend templates*. It has been applied to the analysis of ICU data, but only to a single patient.

The Time Series Workbench provides knowledge-based event detection in complex time series data [34]. High-frequency data is approximated through a series of line-segments. The merging algorithm is in fact an instance of temporal interpolation. It is controlled via a single relative error threshold. A rule-based system is used for pattern matching. The technology does not operate in real time, but is retrospective. The system has been evaluated in a neonatal intensive care unit.

Miksch et al. [44] have developed methods for the abstraction of repeated patterns in high-frequency, rhythmical data (like data from ECG). They have designed an algorithm to transform a curve constituted by a series of data points into a set of bends and lines in between them, similar to the way such graphs appear to domain experts. The resulting qualitative representation of the curve can be expressed as a list of objects each describing a bend. In this format, it can be utilised in a knowledge-based system, or for visual augmentation of the original data.

Seyfang et al. [62] use time-oriented, knowledge-based abstraction methods to optimise oxygen supply for neonates. The system is based on an abstraction called *spread*, which is used to derive steady qualitative descriptions from oscillating high-frequency data using regression lines calculated in a sliding time window [43]. The width of the spread shows the uncertainty involved in its calculation. This abstraction is used as the basis for a knowledge-based process to control the oxygen supply in a minimal invasive way. The system was successfully evaluated in a clinical trial [82].

Charbonnier et al. use a method of segmentation based upon gradual linear approximations for on-line extraction of temporal episodes from ICU high-frequency data [12]. At any time, the latest two line segments are classified into nine possible temporal shapes. These shapes can be summarised into the three major cases *steady*, *decreasing*, or *increasing*. The proposed system provides real-time detection of potentially dangerous developments.

## 2.3 Discussion

The results of applied temporal abstraction techniques are encouraging. The various methods provide useful decision support for the medical domain. However, most methods are tailored to a small spectrum of use cases and the different approaches are usually not integrated in a general framework. The Knowledge-Based Temporal-Abstraction method provides such a general framework. Unfortunately, the developed applications are not suitable for on-line analysis of high-frequency data in intensive care.

Intensive care is an area where there is need for improved decision support and high-frequency processing. Therefore, our aim is to provide tools that are capable of handling high-frequency data as well.

However, temporal abstraction and monitoring are not enough. The integration of guideline execution into the clinical data flow becomes more and more important in order to apply decision support systems to clinical daily practice [32,63]. This allows to go beyond diagnosis and prognosis, and provide computer-implemented treatment support.

There are many guideline modeling approaches today, but only few integrate strong data abstraction resources (compare section 2.1). Most guideline execution systems are database-centric and are not well-suited for high-frequency domains. The limitations are not necessarily inherent to the methodologies, instead they reflect the design goals set by the research groups implementing the systems.

Asbru fills this gap providing strong temporal abstraction capabilities integrated with guideline execution. The language can be used for the temporal abstraction task as well as guideline modeling in diverse domains [61]. Unfortunately, an execution engine that would live up to the potentials of the Asbru language did not exist previously. None of the existing systems closely integrate plan execution and the required data abstraction into the clinical data flow in a high frequency domain such as intensive care (compare section 2.1.1).

For the given reasons we have decided to implement a suitable execution engine for Asbru, the Asbru Interpreter. The theoretical work of Seyfang provides the foundation for this implementation [59].

# Chapter 3

# Introduction to Asbru

This section gives an introduction to the Asbru language. It is mostly a summary of the available resources: The complete syntax of Asbru can be found in the Asbru Reference Manual [60], a more detailed introduction to several aspects of Asbru is available in [41, 42, 45].

## 3.1 The Asbru Plan Library

An Asbru plan library is written in XML[1] and consists of two major parts, the *domain definition* section and the *plans* section. The domain definition defines both quantitative and qualitative parameters (i. e. input data and abstractions of input data).

The plans section contains the plan definitions. Plans can be grouped and hierarchically nested. The plan components, such as conditions or intentions, refer to the parameters defined in the domain definition.

## 3.2 Representation of Patient Data

In Asbru, all patient data and world-state is represented as either *parameters* or *variables*. *Parameters* are time-stamped data and the implementation allows extensive reasoning about past states of a parameter. *Variables* are timeless data, which behave like variables in programming languages. They should only be used for secondary tasks, such as keeping status information.

Parameters are input channels of the system. The data can be retrieved from various data sources, e. g., text files, measuring devices, or a relational database system. The modular architecture allows to easily implement custom modules for specific data sources. This also facilitates the integration with existing patient record systems.

[1] The current DTD for Asbru, documentation, and the reference manual can be found at: `http://www.asgaard.tuwien.ac.at/plan_representation/asbrusyntax.html`. Accessed Feb 26, 2006.

## 3.3 Data Abstraction

Data abstraction deals with the transformation of information obtained from external sources. Most of the data is delivered as quantitative measurements. However medical concepts are usually not expressed in terms of quantitative values but instead as context-dependent, higher-level concepts, such as "high blood pressure".

Data abstraction is most relevant in high-frequency domains, in which sensors deliver input at a rate of several records per minute or second, but it is also applicable to low-frequency domains. The issues of data abstraction fall into three main categories: data validation, calculation of derived values, and transformation into qualitative information [42].

Any data obtained from sensors varies in quality over time. Under good conditions it is fairly reliable and exact, but under circumstances not directly visible to the monitoring process they become very unreliable. Data validation must compensate for spurious values in input data and protect downstream modules from getting confused by invalid data.

In addition to the original data, a number of statistical measures derived from that data are important for the abstraction of higher-level concepts. Regression lines and trend analysis are important examples of derived values. For instance, determining if a parameter is increasing, stable, or decreasing is a common task.

Validated data and derived values still have the form of two-dimensional data points describing the value of a parameter at a certain instant in time. To facilitate reasoning about the observed data, the quantitative values should be transformed into qualitative values, and instantaneous measurements that have the same qualitative values should be concatenated to an interval over time during which the abstracted parameter's value stays unchanged.

Asbru provides support for these three data abstraction categories. Additionally, it provides temporal abstraction capabilities to facilitate reasoning in a time-oriented way. In [61], a more detailed report of Asbru's temporal data abstraction capabilities is given. It is illustrated with two examples from the domains of artificial ventilation and diabetes.

## 3.4 Temporal Patterns

Asbru provides a rich representation to describe the temporal dimension of values to be observed. The basic syntactic construct is the temporal pattern. All conditions for the transition from one plan state to another are expressed as temporal patterns. Temporal patterns can also be used to obtain qualitative information through temporal abstraction. An observation matching a temporal pattern is called an *episode*.

A temporal pattern consists of a parameter proposition, a combination of multiple nested temporal patterns, or a plan-state descriptions. Temporal patterns can be combined using *temporal constraints* (3.4.3) and *constraint combinations* (3.4.4).

### 3.4.1 Parameter Proposition

Parameter propositions are defined using the `parameter-proposition` element and consist of a parameter name, a value description, a context and a time annotation [60]:

**Parameter name.** The name of the parameter to be monitored, e. g., *body-temperature* or *blood-glucose level*.

**Value description.** A predicate or comparison both referring to the named parameter, either a value description, a value range, or a query for its availability, e. g. *higher than* 37 °C or a qualitative value such as *high*.

**Context.** A set of parameters and their allowed values. The context must hold during the whole period of time in question. It is used to describe a set of situations under which the temporal pattern as a whole is valid.

**Time annotation.** A flexible description of the interval during which the designated parameter must take the value given by the value description for the parameter proposition to become true. See section 3.4.2.

The point in time when both the parameter and the context start matching their respective descriptions is called the *positive flank*. The point in time after the positive flank, when one or both, parameter or context, stop matching their descriptions, is called the *negative flank*. If the interval between positive and negative flank matches the *time annotation*, this interval is called a *fitting interval*.

### 3.4.2 Time Annotation

A time annotation describes an interval of time. It specifies three time ranges constraining start, end, and duration of the interval. The time annotation therefore allows a representation of uncertainty in starting time, ending time, and duration [58]. In the Asbru language the constraints are defined as shifts relative to a reference point. This enables to easily define time annotations relative to an event not known at plan creation time, e. g., the start of plan.

As already said, the constraints on start and end of the interval are defined as shifts (offsets) relative to a *reference point*. The *earliest starting shift* and the *latest starting shift* define the *earliest* and the *latest* point in time when a fitting interval must start. The *earliest finishing shift* and the *latest finishing shift* define the *earliest* and the *latest* point in time when a fitting

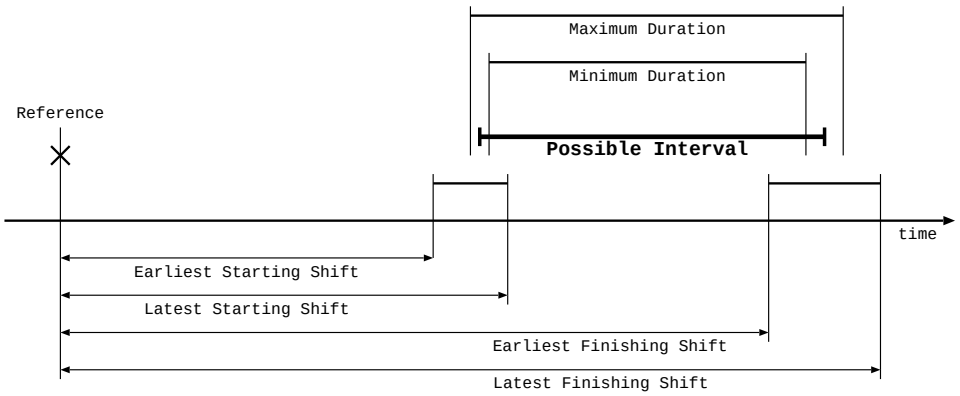


**Figure 3.1:** A schematic illustration of the Asbru time annotation [60]. The figure shows the constraints for the positive flank, negative flank, and the duration, and a possible fitting interval.

interval must end. *Minimum duration* and *maximum duration* constrain the duration of a fitting interval. Figure 3.1 illustrates the time annotation.

The reference point can be either a fixed point in time, the current time (*now*), a reference to a plan-activation or parameter-change, or a set of cyclical time points. The latter three allow defining repeated temporal patterns.

### 3.4.3 Temporal Constraints

Temporal constraints allow to define temporal relations of two temporal patterns using the thirteen temporal relations defined by Allen [2]. Temporal constraints are met if two observed intervals from the two inputs meet the defined relation.

Asbru implements only seven of the thirteen relations. The set of thirteen contains inversions for the relations too, which can easily be accomplished by exchanging the arguments. Temporal constraints are defined using the `temporal-constraint` element. See table 5.3 on page 79 for the list of supported temporal relations.

### 3.4.4 Constraint Combinations

A constraint combination connects two or more temporal patterns using one of the Boolean operators *and*, *or*, or *xor*. The output of a constraint combination is defined as follows:

**and** The constraint combination *and* is fulfilled if (and only if) there is at least one currently valid episode for each input temporal pattern.

**or** The constraint combination *or* is fulfilled if (and only if) there is at least one currently valid episode for at least one input temporal pattern.

**xor** The constraint combination *xor* is fulfilled if (and only if) there is exactly one input temporal pattern with more than zero currently valid episodes.

Constraint combinations are defined using `constraint-combination`. Additional to constraint combinations, Asbru provides an element to define the Boolean negation of a temporal pattern: `constraint-not`.

### 3.4.5 Analysis of Episodes

Episodes matching a temporal pattern can be further analysed to abstract higher-level concepts from a temporal pattern. The `episode-analysis-def` element provides several operators to extract the following features from a stream of episodes matching a temporal pattern:

- The number of currently valid episodes that match the temporal pattern.
- The duration of the latest matching episode.
- The start of the latest matching episode.
- The end of the latest matching episode.
- The total duration of all currently valid episodes.

A count constraint on a temporal pattern (element `count-constraint`) allows to compare the number of episodes that match the temporal pattern with an expression evaluating to a numeric value. The count constraint is fulfilled, if the temporal pattern is fulfilled for as many times as defined.

### 3.4.6 Boolean Representation of Episodes

Another intuitive way to create further abstractions on the result of a parameter proposition is to transform the found episodes into a Boolean parameter. The stream of episodes is translated into a series of intervals during which the Boolean value is true if (and only if) an episode occurs at this particular moment in time. In other words, at the positive flank of an episode a data point with value *true* is issued, at the negative flank one with value *false* [61].

This is done by encapsulating a temporal pattern using the `boolean-def` element. Note that the result is usually only useful for parameter propositions, because e. g., the episodes issued by temporal constraints enclose the input episodes, which means that the Boolean parameter will be true during each convex (enclosing) interval formed by a pair of input episodes.

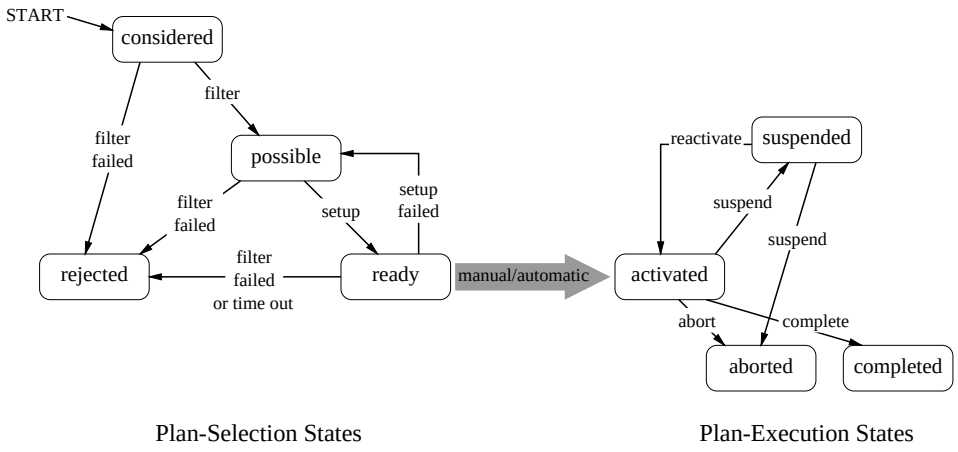


**Figure 3.2:** The Asbru plan states and conditions for plan state transitions [60]. The states of the plan selection phase are shown on the left-hand side, the execution phase states on the right-hand side of the figure. In Asbru 7.2, the activation of a plan can be manual or automatic, in the latter case the state *ready* is skipped. In Asbru 7.3 each state transition can be specified to need user confirmation. Therefore, the ready state of version 7.2 is obsoleted in version 7.3.

## 3.5 Plan States

In the Asbru plan-state model, all plans and actions are durative, therefore, a set of mutually exclusive *plan states* describes the actual state of the plan during both plan selection and execution phase. Conditions specify state transition criteria for the transition between neighbouring plan states [41]. Figure 3.2 illustrates the plan states and the possible transitions based on the transition criteria.

Plan states of parents and children are synchronised through propagation of plan states. If a parent plan is aborted or suspended, this is also propagated to its children. Plan states are also propagated from the children to the parents. There are several ways to control the exact propagation semantics between children and their parent, for example through the *propagation specification* (see [60]).

## 3.6 Components of Asbru Plans

An Asbru Plan consist of a name, a set of arguments, a time annotation representing the temporal scope of the plan, and five components or *knowledge roles*: preferences, intentions, conditions, effects, and a plan body. The arguments, the time annotation, and all components are optional [45].

### 3.6.1 Preferences

Preferences bias or constrain the selection of a specific plan to achieve a given goal and express a kind of behaviour of a plan (strategy, utility).

### 3.6.2 Intentions

Intentions specify high-level goals at various levels. Intentions are temporal patterns of actions or world-states that should be maintained, achieved, or avoided. They are not only used for plan selection, but are important for critiquing to decide if alternatively taken actions (non-compliance with the guideline) have nonetheless fulfilled the intentions of a plan.

Intentions can be grouped in four categories: they can define an intermediate state or an intermediate action as well as an overall state pattern or an overall action pattern.

### 3.6.3 Conditions

Conditions are temporal patterns that need to hold at particular plan steps to induce a particular state transition of the plan instance. There are six different conditions that enable transition from one plan state into another. Two of those are preconditions for plan states before the activation of a plan (filter-precondition and setup-precondition), the other four conditions induce the plan state transitions after the initial activation of a plan (suspend-condition, reactivate-condition, complete-condition, and abort-condition).

### 3.6.4 Effects

Effects describe the functional relationship between parameters resulting from the execution of the plan or the overall effect of a plan by means of qualitative functions. Effects have a likelihood annotation, i. e. the probability of occurrence of the effect.

### 3.6.5 Plan Body

The plan body contains subplans or actions that are to be performed when the plan itself is activated. The plan body might also be a reference to the plan body of another plan (*refer-to*), or the plan might be declared abstract (*to-be-defined*), i. e. only inherited plans will define a plan body. A plan might be a non-decomposable action to be performed by a user (*user-performed*). To define cyclical plans, the *cyclical-plan* element is used in the plan body.

#### Subplans

To decompose a plan into one or more subplans, the *subplans* element is used in the plan body. This element is used to group plan steps in one of

the following temporal orderings:

**sequential** The steps are performed in strict order, each is started only after its predecessor is finished.

**parallel** All steps or subplans are started at the same time, they may finish at any time.

**any-order** The subplans are performed one at a time, without strict ordering, but at each instant in time, only one step is performed (i. e. their execution will not overlap).

**unordered** There is no ordering between the steps. They may overlap or not, and each may start and end whenever appropriate.

Additionally there are several properties that can be defined to control the exact semantics of subplan control. The continuation specification defines the number or the subset of the steps which must be performed before the parent plan can complete successfully. The waiting strategy allows to define if the parent plan should wait for optional subplans. It can also be defined that aborted subplans should be retried. Finally, the propagation specification allows to define exactly when the abortion of a subplan should be propagated to its parent plan (i. e. if the parent plan should be aborted as soon as a specific subplan or a certain number of subplans are aborted or rejected).

# Chapter 4

# Design

This chapter gives an overview of the Asbru Interpreter from a software design point of view. It introduces the basic concepts of the module framework, and how the various components interact with each other. I describe the data flow in the Asbru Interpreter and its implementation in an object oriented language, specifically Java.

Section 4.1 describes the overall architecture of the Asbru Interpreter. Section 4.2 details design decisions for the modules and the framework, and analyses the requirements for various components. In section 4.3, I present the class model of the Asbru Interpreter. Section 4.4 describes the interaction of the different components during execution. Finally, section 4.5 concludes the chapter with an example from the field of artificial ventilation of neonates.

## 4.1 Architecture

Conceptually the Asbru Interpreter consists of three basic units: *data abstraction*, *monitoring*, and *plan execution*. In the data abstraction unit, various temporal or atemporal abstractions are applied to the patient data to gain information at higher conceptual levels. The provided quantitative or qualitative data is monitored to detect temporal patterns in the abstracted data. This information is used to control the selection and execution of plans. This data flow is not unidirectional, instead, the execution unit can interact with both monitoring and abstraction unit to adjust the monitored patterns and to adapt the abstraction process to the context given by the current plan states.

The idea of modularising data abstraction as network of modules goes back to functional programming and control theory. Each module is responsible to compute a single (simple or more complex) transformation of the input data. A module instance is created for each abstraction, e. g., for the product of two inputs, the average of a list of values, or the comparison of

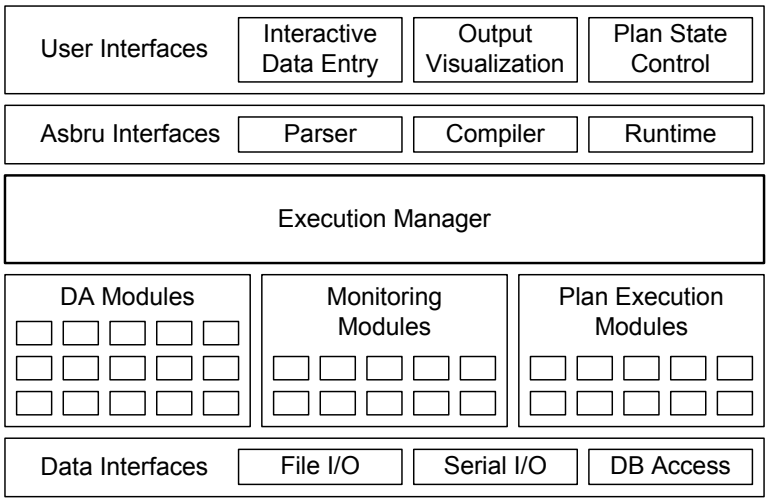


**Figure 4.1:** The architecture of the Asbru Interpreter. The Asbru Interpreter is designed in several layers. Communication between these layers is restricted to well defined APIs.

two values.

Monitoring and plan execution were originally seen as components independent of data abstraction. Nevertheless Seyfang recognises that data abstraction and guideline execution should be integrated [61]. The three conceptual components mentioned above – data abstraction, monitoring, and plan execution – can be seamlessly integrated in a directed graph of modules.

Figure 4.1 shows the architecture of the Asbru interpreter. The Asbru Interpreter is designed in several layers, communication between these layers is restricted to well defined APIs. The Asbru Interface layer contains the components to translate an Asbru plan library from the XML definition into a module graph. Each Asbru element in the plan library is mapped into one or more modules, which are grouped into data abstraction, monitoring and plan execution modules. The Execution Manager controls the data flow between the modules and represents the core execution component. The Data Interface layer provides access to patient data from various sources.

Figure 4.2 shows the data flow in the Asbru Interpreter. At program start, the Asbru plan library XML file is compiled into a directed graph of modules. For each time step, the Execution Manager enacts each of the modules in the network to process patient data, monitor temporal patterns, and execute the plans in the guideline. These modules are largely compatible with each other, which allows information extracted by any module to flow back into the abstraction or monitoring process. To handle complex networks with many inputs in high-frequency applications, the Execution Manager

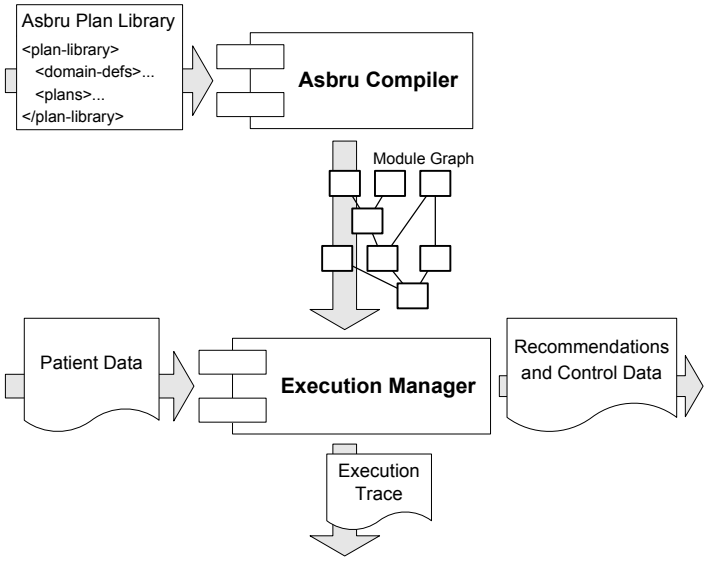


**Figure 4.2:** Data flow in the Asbru Interpreter. The Asbru plan library is compiled into a directed graph of modules by the Asbru Compiler. The Execution Manager uses this module network to process patient data and execute the plans representing the guideline. The output of the modules is provided in a uniform way, to be displayed in a graphical user interface. All state transitions are documented in a log file which for the analysis of the execution path.

ensures that each module is enacted exactly when needed, allowing for small time steps by some modules without the overhead created by other modules which would not provide new information at that moment.

The output of the system can be displayed in a graphical user interface. Using custom-built modules, the output can be integrated with the control of medical devices in a close-loop setting.

During operation, the interpreter writes an extensive log file documenting all abstraction steps and plan state changes. This is later transformed into various reports focussing on different aspects by easily customisable post-processing tools.

More information about the various subsystems and components is provided in the next sections.

## 4.2 Requirements Analysis and Design Decisions

This section discusses functional requirements for important components of the Asbru Interpreter. From these requirements I try to make sensible definitions for the expected behaviour of the various components. I begin with describing the properties of *modules*, the basic processing units in the inter-

| Module | Description |
|---|---|
| AddModule | Add the result of two or more numbers. |
| ConstantModule | Outputs a single constant value. |
| MaximumModule | Outputs the maximum value of all input channels. |
| DayOfWeekModule | Extracts the day of the week from a date. |
| EqualModule | Outputs if the two inputs are equal. |
| AndModule | Outputs the logical *and* of two or more inputs. |
| HistogramModule | Computes a histogram of a series of values. |

**Table 4.1:** Examples of simple Asbru modules.

preter and continue with *data points*, which define how information can be passed between different modules. After describing the requirements for the *Execution Manager*, which is the core component in the execution process, I conclude this section with a discussion about the *playback* mode in the interpreter.

### 4.2.1 Asbru Modules

As explained above, the Asbru Interpreter works by connecting instances of self-contained, independent modules to represent the definitions in the plan library. The base class for all modules is the BaseModule. All modules are derived from this single class and inherit its properties. The requirement for the BaseModule class is that it must provide all outside visible properties and methods for any module. Table 4.1 on page 33 lists some examples of simple modules.

#### Defining Modules

**Input of Data and Output of Results.** Each module must be able to receive input data, calculate the new output and return the result. Each input and output has a time stamp at which the value becomes valid, the *valid time*. A module does not receive an input before the input's valid time. On the other hand it may output a value with a valid time in the future. For example there is a simple DelayModule that delays each data point by some amount of time simply by adding the defined time span to the valid time of the input value.

The method for processing input values is called newData, as proposed by Seyfang [59], and receives a single DataPoint argument. The newData method must be implemented by every module and returns a new DataPoint.

At any time, a module may decide that it does not want to output anything at all for the current input. In this case it returns null.

If a module has no parents in the module graph, it will never receive input

but it must still output data. There are two kinds of modules that may act as root elements in the module graph: *constant modules* and *raw data modules*. Constant modules represent constants found in Asbru plans, these modules are only called once and their output cannot change afterwards. Raw data modules are modules that receive their input from outside of the system, e. g., a module may read data from a file or receive data from a measuring device.

Summarising, we can define an Asbru module as:

**Definition 4.1 (Modules)** *An Asbru* module *is a single processing element in a directed graph that receives input from its direct parents in the graph. A module can have an arbitrary number of children receiving its output. For each time step the module's processing method is called exactly once. If the module has more than one parent, it receives a data point* set *with the input data from all parents. A module may output a single* data point *with a defined* valid time *at each time step.*

**Definition 4.2 (Modules)** *Root modules in the graph (i. e. modules without a parent) must be either constant modules or raw data modules. Raw data modules provide a method to return the next point in time at which they want to output a data point.*

**Alarms.** A module may need to output data at a specified rate, or it may want to be triggered at certain times to change its internal state. This is for example required by one version of the TimeWindowModule that outputs all input values in groups at a specified rate, independent of how many inputs it has actually received during that time.

Therefore, a module can register alarms. For each triggered alarm, the timeout method of the module is called, which receives the valid time of the alarm and a flag that can be defined by the module when registering the alarm in the ExecutionManager. The method may optionally return a new DataPoint.

Alarms are registered for a specified point in time. If both input data and alarms are available for a specific point in time, the sequence of delivery is important. For instance the TimeWindowModule may want to include the input for the current time step (alarm after input), or it may want to exclude the input for the current time step (alarm before input). Therefore, we define *pre-alarms* and *post-alarms*. Both can be registered independently in the ExecutionManager using setPreAlarm and setPostAlarm, respectively.

A module may output data for a single time step at *pre-alarm*, new data input, or *post-alarm*. That raises the question of what happens if a module outputs more than one data point for one time step. More than one output per time step would violate definition 4.1, therefore we define:

**Definition 4.3 (Alarms)** *A module is called for registered alarms. An alarm is delivered to the module* before *the data input for the current time step for* pre-alarms*, or afterwards for* post-alarms*. The module may output a new data point for each alarm.*

*If a module outputs more than one data point for a given valid time, each subsequent data point replaces the previous one. If the data points can be merged*[1]*, they are combined into a single data point instead.*

**Control Inputs.** Many modules have properties that define how exactly data input must be processed. For the TimeWindowModule this may be the length of the window. Most of the time these properties are known at the time of compiling the Asbru library, for instance when they are defined as attributes of the element in the XML file. In those cases the properties can be given to the module at the time of instantiation. Some of the properties can themselves be defined dynamically based on further elements, though. When the Asbru compiler cannot evaluate such a property to a constant expression, it must construct a subgraph of modules to evaluate the property at runtime.

Such a property could be delivered to a module as an additional regular input. This seems unintuitive because these properties must be known before real data input can be processed, and they are usually defined to be fixed as soon as they are set once. To accommodate these differences we define a new data flow channel between modules: *control inputs*.

**Definition 4.4 (Control Inputs)** *A module can have one or more* control inputs*. The output of any regular module can be connected to the control input channel of such a module. Control inputs are seen as regular parents of the modules, with the following differences: A module cannot output a new value when it receives new control input, instead it must only change its internal state. A module can tell the Execution Manager that the control input for itself is frozen and will not change any more during the execution. In this case the manager stops delivering control input changes to the module.*

Control inputs are delivered to the module in the newControlData method receiving a DataPoint. The method must return *true* or *false* to indicate if control input has been frozen. Control inputs are also important for *playback*, described in sections 4.2.4 and 4.4.2.

**Definition of Input and Output Data Type.** A module is able to declare what kind of input it expects and what the output of the module will be. This enables the Execution Manager to check the module graph for inconsistencies and therefore to detect bugs in the Asbru compiler. These properties include:

[1] DataPoints can be merged if the respective class implements the interface MergeableDataPoint and the merge actually succeeds.

- The input data type.
- The cardinality of the input, i. e. how many input channels the module expects, e. g., no input, one input, one or more, two or more inputs.
- The control input data type.
- The cardinality of the control inputs.
- The output type.

### Raw Data Modules

Raw data modules are modules that output data independent of inputs and usually have no parents in the graph. These modules extend the class RawDataModule. As said in definition 4.2, raw data modules must provide a method to return the next point in time at which they want to output data. This method is defined in RawDataModule and is called getNextInternalTime. It returns the time for the next expected output in the internal time of the interpreter.

If a module does not have any more data to output, it must return the symbolic value NO_MORE_DATA. If a module does not know the time for the next data, it may return ASK_AGAIN. This instructs the Execution Manager to query the module again in the next time step.

Since this causes a lot of overhead if the internal frequency of the interpreter is set to a high value (the default) and the output of the raw data module is of lower frequency, it is strongly recommended that the module calculates a good approximation for the next output and returns this to the Execution Manager. The manager will then ask for new data at the specified time.

### Plan Modules

Plan modules are special in that a parent plan needs to send data to its children, propagating plan states and synchronisation information, but the children need to propagate their plan state or a return value back to the parent. This introduces cycles into the module graph, which would be free of cycles otherwise. This means that we cannot propagate inputs in such a way that a module is called only once for each time step, receiving all information.

Therefore, we change the timing of the output of plan modules so that the output at time $t$ will only be delivered to the connected modules at time $t+1$. This breaks the cycles by delaying by one time step any data flow that may go *upwards* in the graph.

Nevertheless this delays plan state transitions and would cause the transitions to fall behind data delivered to plan modules. This is avoided by

dividing the internal steps for regular data flow, from now on called *macro steps*, into an arbitrary number of *micro steps* that are reserved for plan state transitions.

**Definition 4.5 (Macro Steps)** *Macro steps are the time steps used by the Execution Manager to process data flow between regular modules. Data input from raw data modules and pushing this data through the module graph must only happen at macro steps. The macro frequency defines the number of macro steps per time interval.*

**Definition 4.6 (Micro Steps)** *Micro steps are time steps between the macro steps that are reserved for plan state transitions, i. e. propagating of plan modules' outputs. The micro frequency is a multiple of the macro frequency. At micro steps, no other modules except plan modules may output data points.*

The micro frequency $f_{micro}$ is usually set to $1000 \cdot f_{macro}$, but the value can be changed for very deep plan module subgraphs. The Execution Manager checks for plan state transitions that overflow the available micro time steps. In the case of an overflow it throws a PlanStepOverflowException and aborts execution because the behaviour would be indeterministic.

All plan modules are based on AbstractPlanModule and output PlanModuleOutputDataPoints.

### 4.2.2 Data Points

A *data point* in the Asbru Interpreter describes a structured unit that is used to send data from one module to the next. Data points can contain anything from single values to a set of other data points.

All data points have some common properties:

**Valid Time.** The valid time defines the point in time when the data becomes *valid* or *known*. A data point is not delivered to a recipient before its valid time.

**Validity.** This Boolean property defines if the data point represents the value of a parameter (*valid*), or if it represents that the parameter has no known value, i. e. its value is undefined (*invalid*).

Each data point class extends the abstract class DataPoint that implements these properties. See section 4.3.1 for a description of the most important data point classes.

### 4.2.3 Episode Data Points

Section 3.4.2 gave an introduction to temporal patterns, parameter propositions and time annotations in Asbru. In short, we can say that a parameter

proposition monitors whether a parameter meets a value description in a constrained interval of time, for a defined amount of time. The parameter proposition is basically defined by the referenced parameter, a value description and a time annotation.

The point in time when the parameter starts to match the value description is called the *positive flank*, when the parameter stops to match, this is the *negative flank*. An interval of positive and negative flank that matches the time annotation is called a *fitting interval*. The interval during which a *fitting interval* is matching the time annotation is called the *interval of validity*.

For a better understanding let us look at an example. A parameter proposition might by defined as *"high blood pressure for at least 1 hour during the last 6 hours"* [2]. Any measurement of high blood pressure after low or regular blood pressure is a positive flank, a measurement of normal or low blood pressure after high blood pressure is a negative flank. An observation that matches the value description and the time annotation is called an *episode*.

We assume that we measure high blood pressure at 8:00 for the first time, at 13:00 we measure regular blood pressure for the first time after 8:00. So the positive flank is at 8:00, the negative flank is at 13:00. Given the minimum duration of 60 minutes, the parameter proposition first matches at 9:00, which is called the *start of validity*. We have found an *episode*, although the negative flank is still unknown.

At 13:00 we detect the negative flank. The complete interval of positive and negative flank is called a *fitting interval*. At 18:00 (i. e. 12:00 plus six hours) the given episode stops matching the time annotation, which is called the *end of validity*. The interval defined by start and end of validity is the *interval of validity*.

An episode data point must be able to represent all this information. Additionally several modules representing temporal patterns in Asbru perform further combinations or transformations of the output of a parameter proposition. Other modules analyse their output. Therefore, we need a unified output format for temporal data. The following questions summarise the capabilities of the feature abstraction modules deployed on found episodes, i. e. the information for which the monitoring process must provide the basis [59]:

- Was there a fitting interval?
- How many fitting intervals are there?
- What is the duration of each of them?

[2] Exactly: minimum duration is 1 hour, earliest finishing shift is -5 hours, reference point *now*.

- When does a fitting interval start, when does it end?

We can derive the following requirements for the output of parameter propositions:

1. The parameter proposition module must output a data point whenever the parameter proposition becomes fulfilled, at the *start of validity*. In the example we must observe high blood pressure for at least one hour to know that the parameter proposition is fulfilled.

2. The output at the start of validity must include the time when the value of the parameter first met the value description, i. e. the *positive flank*. In the example, when we first observed high blood pressure.

3. The output must include the time when the value of the parameter stopped to meet the value description, i. e. the *negative flank*. If the negative flank is not known at the start of validity, like in our example, then a separate data point must be output with the positive and negative flank as soon as the latter is known (referred to as *end of before-found interval*).

4. If the parameter proposition's output is known to stay unchanged independent of any further input, *end of monitoring* must be reported.

5. An episode must be identifiable uniquely, so the output needs an identifier that is the same for all data points referring to the same episode, but unique across all different episodes.

A parameter proposition module does not output data points for more than one episode at a time. Other temporal patterns may result in overlapping episodes. Because of the way we define a module (definition 4.1), a module must not output more than one data point at a time. Therefore, we must add another requirement for the output of temporal modules:

6. A module must be able to output more than one information about an episode, as well as information about different episodes in a single data point.

To meet these requirements I propose the following properties for an EpisodeDataPoint:

**Start and end of validity.** The start and end of validity matches perfectly with the *valid time* that each data point has. The valid time describes when the information of a data point becomes valid, just as the start of validity describes when an episode becomes valid. We will later see that sometimes an episode is detected earlier than its start of validity.[3]

[3] See section 5.1.5 for parameter propositions with reference point *now*.

If we set the valid time of a data point to a later time, the Execution Manager will only deliver the data point to the subsequent modules at the specified time. These are exactly the correct semantics for the start and end of validity.

**Episode data objects.** A single EpisodeDataPoint may contain zero, one, or more episode data objects (EpisodeInfo), each representing information about a single episode. All episodes share the same valid time, but they have a flag to tell the meaning of each episode information.

**Episode information flag.** Each episode data object has a flag that describes what information the data object provides about the episode. The values for the flag are *start of validity* (SV), *end of validity* (EV), and *end of before-found interval* (EOBFI, i. e. negative flank found). A data object can represent more than one type of information, therefore, the flags can be combined.

This is also useful for merging data point objects. The semantics for the merging of the flags are:

- *SV* merged with *EOBFI* results in only *SV*, the end was not found *"before"*.
- *EV* merged with *EOBFI* results in the combination of both.
- *SV* merged with *EV* means that the episode was never valid (start and end at the same time), and there was no real episode. Depending on the implementation, the complete episode is deleted, or both flags are left as-is. An episode information with *SV* and *EV* set must therefore be ignored by subsequent modules as a possible artefact of the merge implementation.

**Episode identifier.** Each episode data object has an episode identifier that is shared by all episode data objects representing the same episode, but unique across all different episodes during the runtime of the interpreter.

**Positive flank.** Each episode data object representing *SV* and *EOBFI* contains the positive flank of the fitting interval, an episode data object representing only *EV* may not contain the positive flank.

**Negative flank.** An episode data object representing *SV* must contain the negative flank if it is already known, for *EOBFI* the negative flank must always be set, for *EV* it may be set.

**Revocable episodes.** If a monitoring module knows that an episode will not have an end of validity, this information can be used for further optimisations. Therefore, each episode data object

contains the information if the episode may have an end of validity (*revocable*) or if its validity is unconstrained (*not revocable*).

**End of monitoring.** The episode data point contains a flag to indicate whether *end of monitoring* has been reached for this module, i. e. the output for this instance will not change any more. This state is global for this instance and independent of a single episode and is therefore contained in the episode data point itself.

A module must not output any more episode data points after it has output *end of monitoring*.

Another important requirement for a sequence of episode data points is that the information of each episode must always be complete, i. e. for each *start of validity* there must be an *end of validity* if the episode's validity is not unconstrained, and for each *start of validity* with an unknown negative flank there must be an *end of before-found interval* report, even if the latter is only known after the *end of validity*.

### 4.2.4 The Execution Manager

The Execution Manager is the core component in the execution process and represents a messaging system that moves data points between modules in the module graph. It must accomplish the following tasks:

- Maintain the module graph.
- Maintain the internal time during runtime and optionally synchronise it with an external time source (e. g., the clock of the computer where the interpreter runs).
- Maintain a list of alarms registered by modules.
- Determine the time steps at which data points must be processed.
- For each time step with new input data or alarms set, process new data for all modules and push output data points to the subsequent modules.
- Where required, keep a record of data points to process at a later time. Whenever previous data must be analysed based on new parameters, provide means to reprocess the recorded data points.

The next subsections describe each of these items in detail.

### Maintaining the Module Graph

The Execution Manager provides methods to add modules to the module graph and define the modules' *parents* (modules providing output) and *children* (modules reading output). The list of modules is ordered in such a way that each parent is ordered before its children. For the propagation of data point through the module graph, the modules must only be processed in order of the sorted list, so that each module is processed before its children.

### Alarms

A module may register alarms for certain points in time to take specified actions (see Alarms in section 4.2.1). The manager must provide methods to register *pre-alarms* and *post-alarms*. For pre-alarms, the module's timeout method is called just before input data for the current time step is passed to the module, for post-alarms, timeout is called just afterwards.

### Determining the Next Execution Time Step

The next time step relevant for execution must be determined by the manager by looking at:

- The next alarms registered by modules.
- Available input data from *raw data modules* (representing external data sources).
- Data points earlier queued for a later time step in the execution process.

The earliest point in time of these three sources is used for the next execution time step.

### Processing each Time Step

For each time step in the execution process the manager must process alarms and new input data for each module. The module list is iterated in the order determined by the module graph with parents before their children. For each module the sequence of tasks is:

1. Process new *control inputs* if the module did not set control inputs to frozen (see section 4.2.1 on page 35).
2. Process all *pre-alarms* available for the module at this time step.
3. If available, process new input data points using the module's newData method. For *raw data modules* request new output from the module.
4. Process all *post-alarms* available for the module at this time step.

5. Store the combined output of alarms and data processing of the module as the current (or future) output of the module (see also module definition 4.3 on page 34).

### Analysing Historical Data (Playback)

The aim of the interpreter is to analyse and monitor data on-line as it arrives. We must admit that there are cases for which this is not possible. Consider an example of the suspend condition of an Asbru plan. The condition is fulfilled if *"the patient had fever for at least 6 hours, at the earliest four days before the plan was started"*. Here, the start of the plan is a parameter affecting the monitoring of another parameter (*blood pressure*), but the start of the plan is only known after the value of the blood pressure was examined.

We see in this example that the interpreter must be able to re-analyse recorded data at a later time. To minimise the effect on regular processing and to avoid duplication of effort, this should be done in a way so that the analysis of recorded data uses the same infrastructure as on-line data processing. Therefore, let me introduce a new mode in the manager, called *playback*.

In *playback* mode the manager processes recorded data again in a subgraph of the module graph. Regular modules are not aware of the difference between on-line and playback mode. Playback can only happen for modules that are currently paused and reset, otherwise a module would have to be aware of the playback operation. A module must be put into this paused state before the start of execution using the manager method variants of setPlaybackTriggers. The manager then sets the module to state paused and determines, which inputs of the module must be recorded for a later playback of the data.

Using one of the setPlaybackTriggers methods we define selected plan states of plan modules that will trigger the playback of a module (or probably a set of modules). This can be done by the Asbru compiler when reading the guideline or at runtime by a plan module itself. Additionally, plan states can be defined to trigger the reset and pausing of modules. If the triggers are to be set while processing data, the compiler must at least activate a module for playback before processing input data.

The only modules that must be aware of playback mode are plan modules. During playback mode plan modules receive the output of the reprocessed data, usually from monitoring modules. The plan modules must change their state according to the new input, but must not yet propagate any plan state transitions. The recorded data is historical and could cause interim state changes that are not correct at the time of playback any more.

After the playback of the recorded data is completed, the participating modules are put into on-line processing mode and the manager proceeds with the next time step after the one triggering playback. At that time, plan state

transitions resulting from the processed data will be performed.

Playback must happen between two regular time steps. This requires that playback is very fast and fits into the time slot available between two regular time steps. For on-line processing of high frequency data this time slot might be very short. I discuss methods to minimise the amount of work to be done at playback in section 4.4.2.

## 4.3 Class Model

In this section I give an overview of the class model of the Asbru Interpreter, with an emphasis on the parts relevant to monitoring temporal patterns. For the purpose of this section, I divide the classes of the interpreter into three major groups:

**Data points.** Data points represent different kinds of structured data to be passed between modules.

**Modules.** Modules are used to represent different elements of the Asbru language. The different kinds of modules are grouped in packages according to their tasks and properties.

**Framework.** Core classes like the manager, the compiler classes, and many utility classes represent the framework.

For each group I show how the defined requirements and the designed model have been translated into a class model suitable for implementation in the object oriented programming language Java 2.

### 4.3.1 Data Points

Figure 4.3 on page 45 shows the class model of the data point classes. The figure only shows the relevant information and hides accessor methods for instance fields.

All data points are inherited from the class DataPoint, which defines the valid time of a data point instance. For supporting valid time and transaction time separately, an additional field must be added for the transaction time. The valid time should then probably be stored as an offset to transaction time to reduce the memory footprint.

The time in a data point is stored in internal time, which usually has at least microsecond resolution. Therefore, time is stored as 64 bit long in the interpreter.

The base class also contains two abstract methods that must be implemented by all inheriting classes: isValid and copy. The isValid method must return whether the data point represents an actual value (i. e. a *valid* data point) or the lack of a value for a parameter (i. e. *invalid* or *unknown* data).

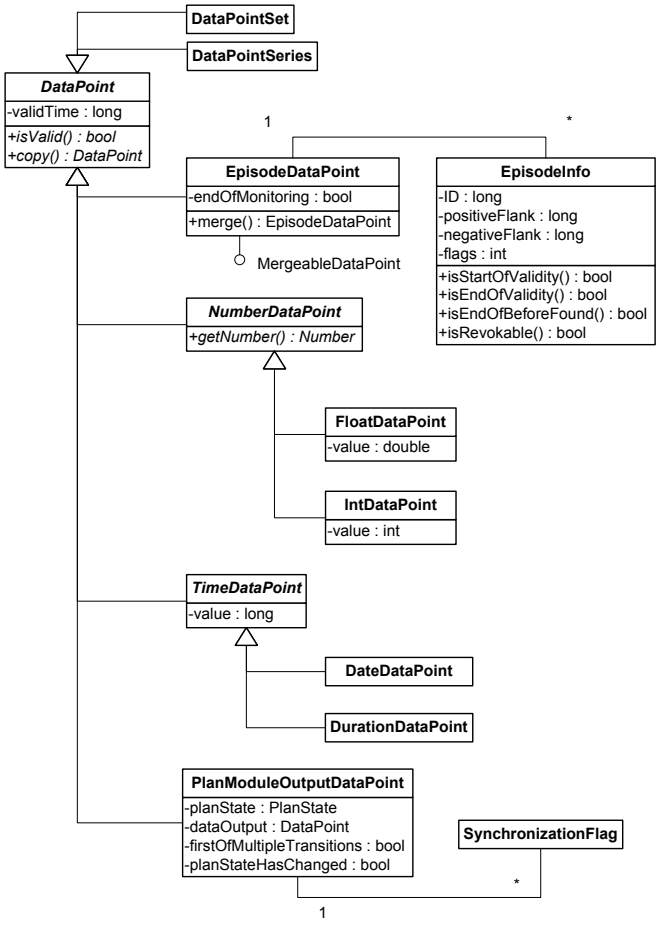


**Figure 4.3:** UML class model of the DataPoint classes.

The validity is mostly stored as a special value in data point classes, e. g., a FloatDataPoint uses $NaN$[4] for an invalid data point, whereas an IntDataPoint uses $-1$ for an unknown value because an integer value cannot be negative[5].

The general contract for any data point class is that all fields must be final and changing the fields of a data point must not be possible. This is necessary so that a data point can be passed around by reference without the possibility of any component changing its property while others use the data point (compare to Java strings).

At some places in the interpreter it is still necessary to create a duplicate of an arbitrary data point with a new valid time. Therefore, each data point

[4] Abbreviation of *"Not a Number"*

[5] An integer value actually represents a qualitative value that is mapped to a positive number.

class is required to implement the copy method that takes a new valid time as argument and creates an exact copy of the data point but with this new valid time.

### Data Point Sets and Series

Whenever a module receives input from more than one parent module, the manager combines the input data points to a DataPointSet. A data point set represents an ordered list of data points. The index of the contained data points corresponds to the indexes of the input modules of the module receiving the data point set.

A data point set itself and each contained data point have their own valid time. A data point set may also contain data points of different types. In Java 5 the DataPointSet could be perfectly implemented using a generic, but because the interpreter is currently targeted for the Java 2 language specification, there exists a separated inherited DataPointSet class for each data point type, e. g., FloatDataPointSet or EpisodeDataPointSet, for convenience.

The DataPointSeries class on the other hand represents an ordered series of values of a specific data type. Such a data point only has a single valid time. The series may have an ordering property attached to it, defining the order of the contained elements. Data point series are output by a single module and treated like any other regular data point. They can be put into sets, of course.

### Number Data Points

Classes inheriting NumberDataPoint are data point types that can be represented as a single number (in addition to the valid time). The FloatDataPoint contains a single floating point number, actually of type double for increased precision.

The IntDataPoint class stores a positive integer number, but really represents a qualitative value in Asbru, such as *low*, *medium*, and *high*, or *true* and *false*. A qualitative value map can be associated with the data point that maps the integer value to the qualitative value represented as text string.

### Time Data Points

Time data points inherit from the TimeDataPoint class. All time values in the interpreter are stored as long values representing the internal time of the interpreter. The internal time starts with value zero at the start time defined in an interpreter project file. The time resolution is also configurable so that a value of 1000 could actually mean 1000 nanoseconds or 1000 seconds after the start of the interpreter. The Execution Manager can convert between the internal time and a regular Java time representation, so it is neither necessary nor advisable to calculate in internal time units directly.

There is a further distinction between a point in time and an interval or duration of time. A point in time should be displayed as a combination of a date and a time component, whereas a duration of time should be displayed e. g., as number of years, months, and days or just as microseconds, depending on the resolution of the value. Therefore, two different classes exist for time representation, the DateDataPoint class and the DurationDataPoint class.

It is also important to note, that a duration of time cannot be accurately stored as just a single number of microseconds. Not every year has the same number of days, because of leap years; not every month has the same number of days; not every day has the same number of days, because of daylight saving time zone changes; and not every minute has the same number of seconds, because of leap seconds. For calendar oriented applications, storing a duration would require four separate fields, one for months, one for days, one for minutes and one for the smallest unit, e. g. microseconds.

### Episode Data Points

Episode data points were already discussed in detail in section 4.2.3. They are implemented in the class EpisodeDataPoint with an inner class EpisodeInfo, where one instance of EpisodeDataPoint can have zero, one, or more instances of EpisodeInfo.

The start and end of validity of an episode is mapped to the valid time of the data point itself, whereas *identifier*, *positive flank*, and *negative flank* are stored as ID, positiveFlank, and negativeFlank in an EpisodeInfo data object. The type of event represented by the data object is stored as a single integer flag using a single bit for each state. It can be extracted using the accessor methods like isStartOfValidity. The latter returns *true* if the EpisodeInfo stands for *start of validity* at the time given by the valid time of the data point. A single data object may represent more than one event (see page 40 for an extensive discussion).

Episode data points implement the MergeableDataPoint interface, so data points with the same valid time can be merged into a single data point.

### Plan Module Data Points

The output of a plan module contains multiple elements of data for several recipients. The PlanModuleOutputDataPoint combines all the elements into a single, structured data point. Such a data point contains:

**Plan state** The current plan state of the plan module.

**Data output** An arbitrary data point that is used for instance in the DataItemModule to set the value of a variable.

**Plan state transition flags** The Boolean flag planStateHasChanged states if the plan module has just gone through a plan state transition and the plan state has changed since the last output of the plan module. The Boolean flag firstOfMultipleTransitions tells if this plan state transition is only the first of two or more plan state transitions happening in same macro time step.

**Synchronisation flags** The data point contains one synchronisation flag for each child of the plan module. The flags are used to initialise, activate, or reset child plans.

### 4.3.2 Modules

Figure 4.4 on page 49 shows the class model of some examples of module classes and the core components of the framework. As above, the figure hides irrelevant fields and members for a better overview.

The base class for any Asbru module is BaseModule. It provides the basic interface needed by the Execution Manager to interact with the modules. The method initialize is called by the manager to give the module a reference to the manager, the method reset is used to reset the internal state of a module. The other methods newData, newControlData, and timeout are called to deliver regular input, control input, and alarms to the module, and receive the output.

Additionally there are two abstract module classes that are special to the manager. Those are RawDataModule and AbstractPlanModule.

#### Raw Data Modules

Raw data modules provide an additional method getNextInternalTime that returns the time stamp of the next available input data. The manager uses this method to determine the next internal time step when new input data will be available. If the time of the next input data is not yet known, a symbolic value can be returned to indicate that the module should be polled again later.

Each module is associated with a DataImport class that provides an interface to the external data source. A data importer may simply read a text file, but it can also interface with an external measuring device.

One DataImport object may provide more than one data channel. A raw data module selects from the channels by either index or name. The channel names are defined in the Asbru library, otherwise they are equal to the name of the parameter defined by the raw data module.

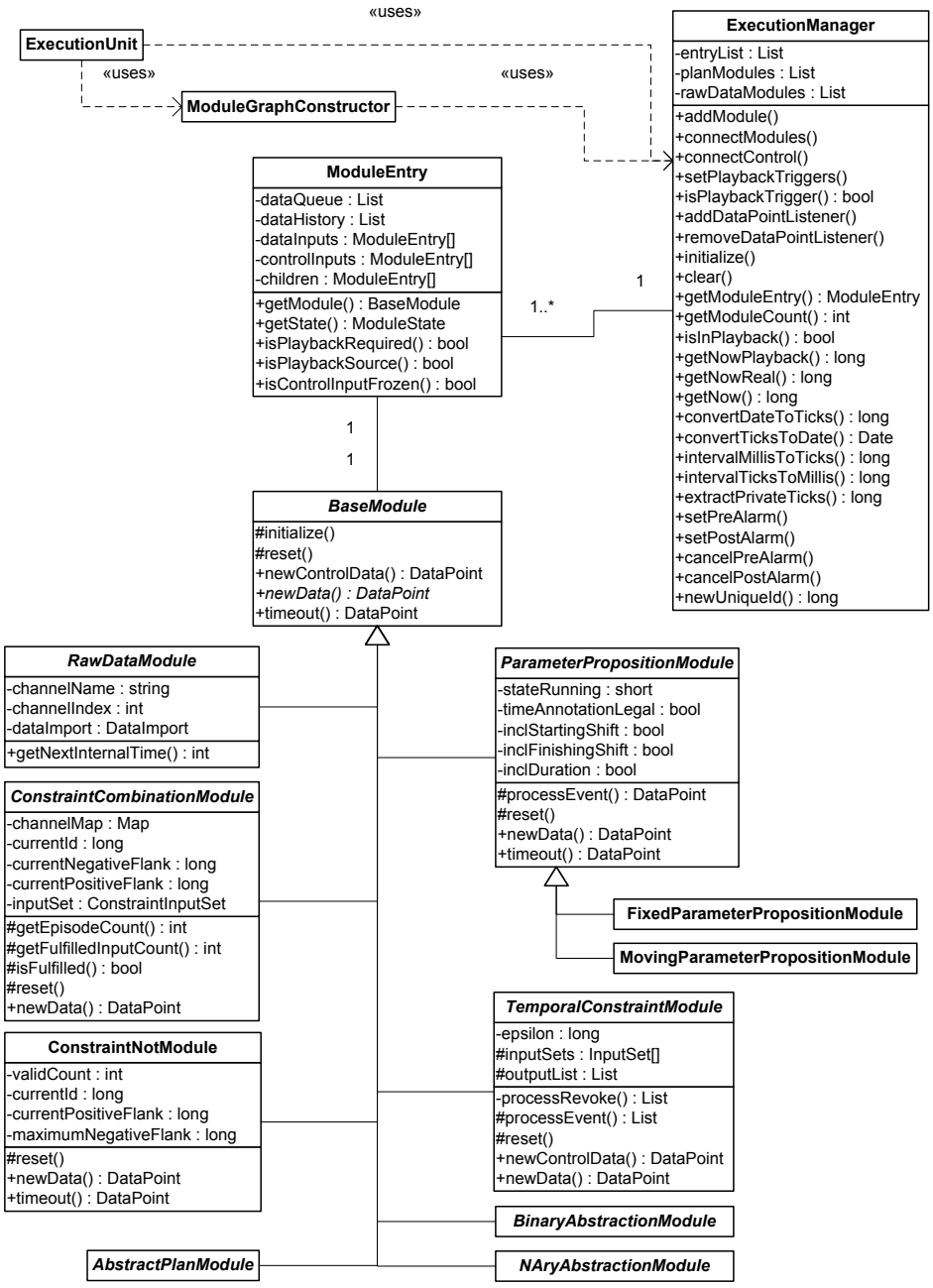


**Figure 4.4:** UML class model of the Asbru Interpreter showing examples of modules described in this work.
Abstract classes or methods are in italic letters, the prefix − stands for *private*, # for *protected*, and + for *public* fields or methods. Not all fields and methods are displayed, method arguments are hidden.

### Plan Modules

Plan modules are inherited from AbstractPlanModule, which implements the basic infrastructure for plan condition checks, plan state transitions, and synchronisation with child plans.

Plan modules differ from other modules in their awareness of the current state of execution and the modules they are connected to. There are some important rules for plan modules:

- A plan module must check whether execution is in on-line or playback mode and must not create any output during playback mode. During playback we must not propagate plan state transitions since we are dealing with recorded data and obviously cannot change the history of the outside world by plan state transitions.
- A plan module is not reset to its initial state at start of playback mode, instead the plan module is itself responsible to manage its state according to the current mode of operation.
- The output of a plan module must always be of type PlanModuleOutputDataPoint. The output at time $t$ must have a valid time of $t + 1$ (see also section 4.2.1).

### Abstraction Modules

Abstraction modules, like the ones given in Table 4.1 on page 33, are usually inherited from either BaseModule directly, or are inherited from e. g., BinaryAbstractionModule for binary operators, or NAryAbstractionModule for operations with any number of inputs.

Most of the abstraction modules are stateless and only work on the current input each time they are called. Others like the TimeWindowModule keep a limited history of data. Such modules must implement the reset method to reset their internal state.

### Monitoring Modules

An important part of my work was to create modules for the *monitoring unit* of the interpreter. Chapter 5 describes the algorithms used for each module. I give a summary of the implemented modules here.

**Parameter propositions.** Parameter propositions are defined using the element `parameter-proposition`. There are three primary types of parameter propositions, those with fixed reference point, reference point *now*, or repeated reference points. The base class for all parameter propositions is ParameterPropositionModule.

The type of the parameter proposition is defined by the reference point of the time annotation. For a time annotation with reference `now`, a MovingParameterPropositionModule is used, for a time annotation with a single fixed reference point a FixedParameterPropositionModule is created. For repeated reference points a RepeatedParameterPropositionModule must be created.

The value description and the context are combined using a Boolean *"and"* to create a single Boolean input for the parameter proposition. The constraining intervals of the time annotation can be specified using the constructor of the module classes or using control inputs at runtime. The output of a parameter proposition is an EpisodeDataPoint.

**Temporal constraints.** Temporal constraints compare the intervals of two episodes using the interval relations by Allen [2]. They are defined using the element `temporal-constraint`. The implementation class is selected, based on the `type` attribute of `temporal-constraint`, the possible values are listed in Table 4.2. The base class of all those implementing modules is TemporalConstraintModule.

| Constraint | Implementing Class |
|---|---|
| before | EpisodeBeforeModule |
| meets | EpisodeMeetsModule |
| overlaps | EpisodeOverlapsModule |
| starts | EpisodeStartsModule |
| during | EpisodeDuringModule |
| finishes | EpisodeFinishesModule |
| equal | EpisodeEqualModule |

**Table 4.2:** The temporal constraint modules.

Input and output of temporal constraint modules are of type EpisodeDataPoint. A temporal constraint takes exactly two inputs.

**Constraint combinations.** Boolean combinations of the results of temporal patterns can be done using `constraint-combination`. The combination is selected with the `type` attribute. The possible combinations are listed in Table 4.3. The base class of the implementation classes is ConstraintCombinationModule.

All constraint combination modules take one or more inputs of type EpisodeDataPoint and output the same type.

**Boolean negation of constraints.** The Boolean negation of a temporal pattern is defined by the element `constraint-not`. This element is

| Combination | Implementing Class |
|---|---|
| and | ConstraintAndModule |
| or | ConstraintOrModule |
| xor | ConstraintXorModule |

**Table 4.3:** The constraint combination modules.

implemented in the ConstraintNotModule. The module takes one input of type EpisodeDataPoint and outputs data points of the same type.

**Episode analysis.** There are some elements in Asbru that analyse episodes output by modules monitoring temporal patterns, foremost the element `episode-analysis-def`. This element has several operators, with each implemented in its own module. Table 4.4 lists the operators with the implementation classes.

| Operator | Implementing Class |
|---|---|
| count | CountModule |
| duration | EpisodeDurationModule |
| total-duration | TotalEpisodeDurationModule |
| start | EpisodeStartModule |
| end | EpisodeEndModule |

**Table 4.4:** The episode analysis modules for the different operators.

The CountModule class is additionally used to implement the element `count-constraint` by comparing the output of the count module with the expression defined in the count constraint, using a regular comparison module.

All these modules process inputs of type EpisodeDataPoint. The output is either an integer number or a time value.

### 4.3.3 Framework

Besides the subset of modules, figure 4.4 on page 49 also shows four classes of the framework. The class ExecutionUnit is the main class used to start the interpreter. The Asbru compiler is implemented in the class ModuleGraphConstructor. As the name of the class suggests, the compiler class constructs the module graph from the plan library, using the ExecutionManager.

#### The Execution Manager

The module graph is maintained in the ExecutionManager class using the ModuleEntry class and a list of entries. The manager creates one ModuleEntry

for each module. An entry object stores references to all regular parent modules (dataInputs), parent modules used as control inputs (controlInputs), and child modules (children).

The entry object also stores the current output of the module and output data points with a future valid time in the dataQueue. If the module is used as a source for playback of another module, the dataHistory list is used to record all output. Additionally it manages all state information associated with a module that is not directly implemented in the BaseModule, because this is irrelevant to the module itself.

The module entries are stored in three lists. The entryList object contains all modules in the graph. The lists planModules and rawDataModules only contain plan modules and raw data modules, respectively.

**Connecting modules.** A module must first be added to the manager using the addModule method. Then the module can be connected to other modules using the variants of connectModules and connectControl to define the parent-child relationships.

**Defining playback triggers.** Whenever the Asbru compiler determines that playback is needed to evaluate an expression in the plan library, it uses the setPlaybackTriggers method to register the plan states of a certain plan module to trigger playback for the corresponding modules at runtime. The method isPlaybackTrigger returns *true* if a combination of plan module and plan state is registered to trigger playback.

**Time lines in the Execution Manager.** The manager has two different time lines, the on-line or real[6] time line and the playback time line. The playback time line is only active during playback. The method getNowReal always returns the current on-line time, the method getNowPlayback returns the current playback time if playback is active. The getNow method returns either of both depending on playback being active or not. So a module that is not aware of playback (i. e. all modules except plan modules) can just use getNow to get the current time step that is processed. Plan modules must always use getNowReal for any output.

The method isInPlayback returns *true* if the manager is currently in playback mode.

**Converting to and from internal time.** The methods convertDataToTicks, convertTicksToDate, intervalMillisToTicks, and intervalTicksToMillis are used to convert dates and intervals (i. e. durations) to and from internal time. The method extractMicroTicks extracts the micro time steps (reserved

[6] The term *real time* might be misleading because this time value must not necessarily be synchronised with the physical clock in the computer, e. g., in batch-mode processing.

for plan state transitions) from a value in internal time as offset to the preceding macro step.

**Alarms.** The methods setPreAlarm, setPostAlarm, cancelPreAlarm, and cancelPostAlarm allow modules to set or cancel alarms from modules. For each alarm, the module's timeout method is called.

**Unique identifiers.** The method newUniqueId returns a new unique identifier of type long that is unique during the runtime of the interpreter, as long as not more than $2^{63}$ identifiers are requested. This method is used in monitoring modules to create unique identifiers for EpisodeDataPoints.

## 4.4 Running the Interpreter

After having given a static picture of the components of the interpreter in the last section, the first part of this section aims at providing an outline of the execution process. A part of a guideline from the field of ventilation of neonates is used as an example. The second part focuses on the problems and solutions associated with re-analysing previous data using the playback mode in the Execution Manager.

### 4.4.1 Outline of Execution

Figure 4.6 on page 56 presents an UML sequence chart of an interpreter run using a very simple plan library that does not contain any plans but only defines a parameter in the domain definition part of the library. Figure 4.5 shows the XML plan library.

The domain definition contains a parameter definition that compares values read from e. g. a file, with a constant expression, and checks if the read value is greater than the constant. In this example the body temperature of a patient is compared to the value 37 °C.

#### Compiling the Module Graph

The process is started by calling the run method of ExecutionUnit. The execution unit reads the project specification, creates DataImport instances to read external data and creates a ModuleGraphConstructor instance, the Asbru compiler. The compiler parses the Asbru plan library using an XML parser library and evaluates the definitions.

To evaluate the parameter $A$ defined in the library, three modules are needed: a FloatRawDataModule to read the value from the source defined in the project file, a FloatConstantModule to supply the constant value, and a GreaterModule to compute the comparison operator *"greater-than"*.

```
<plan-library>
 <domain-defs>
  <domain name="example">
   <parameter-group>
    <parameter-def name="body-temp" type="temperature">
     <raw-data-def mode="manual"/>
    </parameter-def>
    <parameter-def name="fever" type="boolean">
     <comparison-def operator="greater-than">
      <left-hand-parameter>
       <parameter-ref name="body-temp">
      </left-hand-parameter>
      <right-hand-parameter>
       <numerical-constant value="37" unit="C"/>
      </right-hand-parameter>
     </comparison-def>
    </parameter-def>
   </parameter-group>
  </domain>
 </domain-defs>
 <plans/>
</plan-library>
```

**Figure 4.5:** Asbru plan library used for the UML sequence chart in figure 4.6.

The compiler creates instances of these modules and adds them to the manager. Then it connects the raw data module as first input and the constant module as second input to the comparison module.

After the compiler has finished, the execution unit calls the initialize method of the ExecutionManager class. The manager now sorts the list of modules according to the dependencies in the module graph and initialises the modules passing the reference to the manager itself and the index of the module in the list, the latter mainly for debugging purposes.

### Execution

The execution starts with calling the run method of the Execution Manager. The execution unit passes the end date and time for the execution as defined in the project file.[7] The run method iteratively calls the step method until the end date is reached, or no more data is available and the execution is finished.

At first the manager executes step for internal time zero. We assume

[7] The end date is especially important when processing off-line data and the plan library contains iterating constructs but the execution should be aborted after a few iterations.

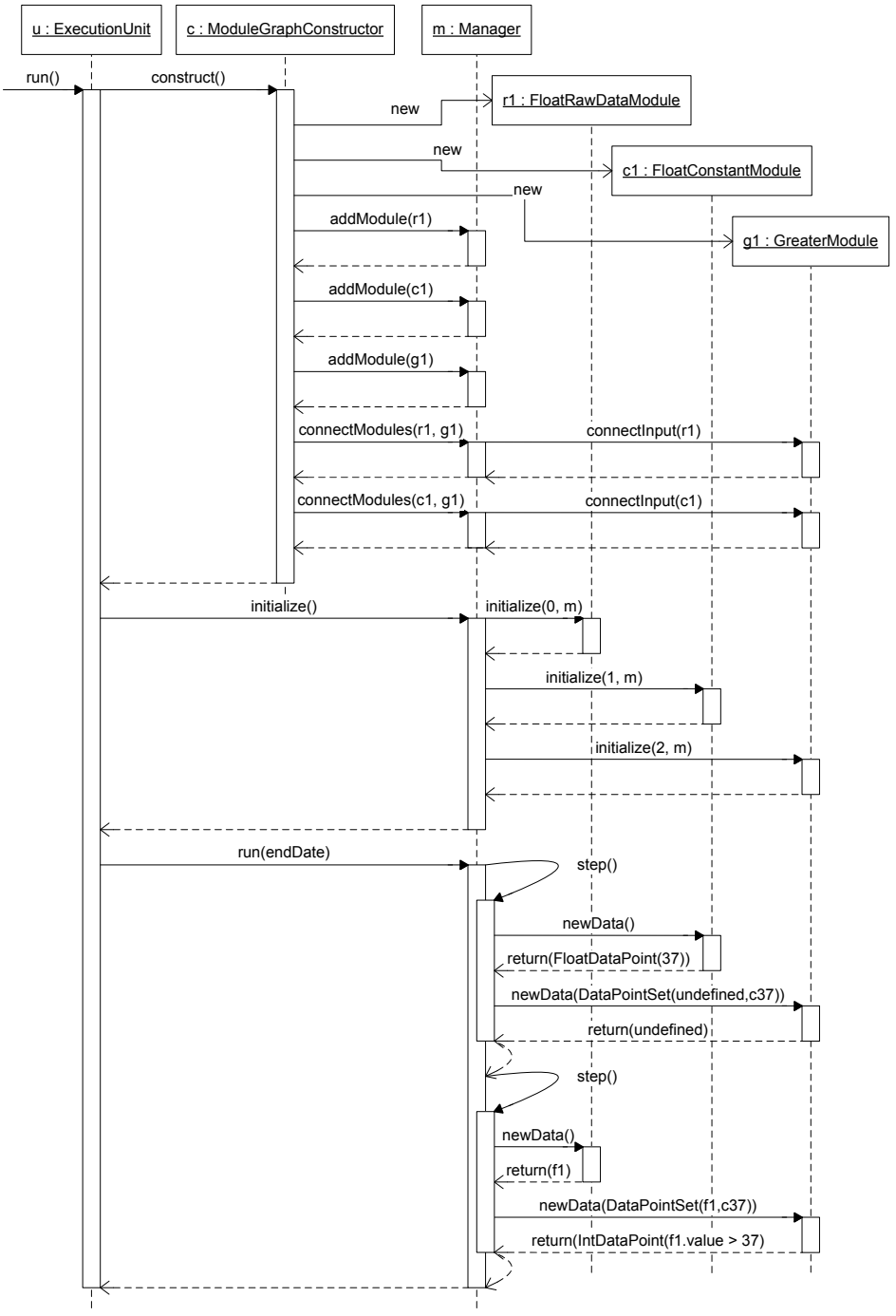


**Figure 4.6:** UML sequence chart of Asbru execution. The diagram shows a minimal example containing no plans but only a parameter declaration in the domain definition (see Figure 4.5). The execution of a plan library including plans is not fundamentally different.

that no data input is available from the raw data module at this time, so the manager only evaluates the constant modules. The manager calls newData of the FloatConstantModule instance c1 and stores the returned data point as the current (and only) output of the module. It then determines that there is new input data for the GreaterModule and processes newData of that module. The input from the raw data module is still undefined, so the output of the GreaterModule is also *undefined*.

The next call to step skips to the first available input data of the raw data module. The raw data module returns the read value as FloatDataPoint, here called f1. The manager again calls newData of the GreaterModule, which is now able to compare the value of f1 with the value of the constant, here 37. The Boolean result is returned as qualitative IntDataPoint.

We assume that there are no more input values available, so the run method returns and the execution is finished.

### 4.4.2 Playback Mode

The *playback mode* has been introduced in section 4.2.4. Here I focus on the Execution Manager's role, the exact semantics, and possible optimisation techniques. In *playback mode* the Execution Manager reprocesses recorded data in a subgraph of the module graph to analyse data with parameters unavailable before. Regular modules are not aware of the difference between on-line and playback mode, only plan modules must adapt their behaviour as explained in section 4.3.2.

The most common use-case for playback is either a parameter proposition with a time annotation containing non-constant shifts or a non-constant reference point, e. g., referencing the start of a plan, or a subgraph of modules processing temporal patterns depending on such a module. In such a case the input data for the parameter proposition must be recorded until the non-constant part of the time annotation is known and fixed. Only then the original input data of the module can be re-analysed.

The sequence of events during playback at time $t_{real} = T$ without any optimisations is:

- At time $t_{real} = T$ the manager determines that playback is required to make the output of module $M$ available.
- The manager switches into playback mode and initialises playback of module $M$ and any plan modules that receive the output of $M$.
- The manager starts at time $t_{playback} = 0$ and first delivers the latest control input data for each module with valid time $T$. This is the control data that was missing before. Using time $T$ makes sure that the modules receive the necessary parameters valid at the current real time, e. g., the start of a referenced plan.

- All time steps for which either input data or registered alarms for module $M$ exist are processed as long as $t_{playback} \leq T$. The processing works just as in on-line mode, with the exception that plan modules do not execute plan state transitions.
- As soon as execution reaches time $T$ again, the manager switches back from playback mode and processes time step $t_{real} = T+1$. At this time step, plan modules are now allowed to propagate plan state transitions that may result from the output of module $M$.
- Module $M$ is switched to on-line processing and is now part of the regular module graph.

The implementation of switching between regular and playback mode is done in the incrementNow method of the ExecutionManager class. The methods initializePlayback and finishPlayback are used to set-up and tear-down playback. The ModuleEntry method getCurrentData returns historical instead of current data in playback mode. Based on playback being active or not, the manager's step method either selects a playback subgraph or the full module graph for processing.

### Optimising Playback

The problem with playback is that data needs to be processed between two regular execution steps, and the available amount of time is very limited for high frequency domains. Therefore, the amount of data that needs to be re-analyse should be minimised, otherwise playback will disrupt on-line processing.

There are basically two ways to reduce the amount of processed data within the given framework:

1. At start of playback use knowledge about the participating modules to select the latest possible start time for playback that will give the same result as starting at time zero.
2. Reduce the number of data points recorded as input for a module to those that represent a qualitative change for the module. For instance, there is no need to store repeated equal input values for a parameter proposition, only changes from *true* to *false* or from *false* to *true* are relevant.

The second optimisation can be implemented quite easily using a filter module in front of the corresponding module. For parameter propositions this is implemented as IntRedundancyFilterModule, which simply filters any input that does not represent a qualitative change in that context. The manager stores the filtered output instead of the output of the original parent module.

The first optimisation is more complex to achieve and is currently not implemented, nevertheless achievable. The heuristics to calculate the earliest required input data should supposedly go into the modules themselves. At least for many common time annotations with a defined earliest starting time or a maximum duration, the maximum data delay can be defined.

The manager would have to deliver the control inputs and query the modules for the earliest required start time. To get the last qualitative state before that point in time, the manager would select the last data point before each module's start time as the first data point to be delivered for the corresponding module. The global start time would be the minimum of those data points' valid time.

## 4.5 Execution of a Real-World Guideline: Ventilation of Neonates

The example above already covered a lot of the manager's interaction with the modules. Of course each step requires much more work than what was shown here, but the aim was to provide a comprehensible overview of the data flow during execution.

From the field of artificial ventilation of neonates, I use for display purposes the following fragment of a protocol controlling the fraction of inspired oxygen based on measurements of partial pressure on oxygen in blood [62].

> "An external monitoring device measures the saturation of oxygen in blood ($SpO_2$). It delivers numeric values at a rate of 1 Hz. These values are abstracted to qualitative values. For example, $SpO_2$ below 80% is mapped to *acute hypoxy*, while higher values are mapped to decreased, normal and increased. If the qualitative value of $SpO_2$ equals acute hypoxy for at least 4 seconds, then normal ventilation should be suspended. In this situation, the patient will receive emergency treatment by the medical staff. If the patient returns to less critical state, as defined by $SpO_2$ being unequal acute hypoxy for at least 10 seconds, normal ventilation is resumed."

In Asbru, *SpO2* is a raw parameter. In addition, we introduce an abstracted qualitative parameter *SpO2-qualitative* with the possible values *acute hypoxy*, *decreased*, *normal*, and *increased*, where *acute hypoxy* corresponds to $SpO2 < 80\%$. Furthermore, a plan called *"Normal Ventilation"* is created with a suspend condition and a re-activate condition, both implemented as parameter-propositions. Each parameter proposition has reference point *now*. The first parameter-propositon has the condition *SpO2-qualitative equal acute hypoxy* and the time annotation *minimum duration* =

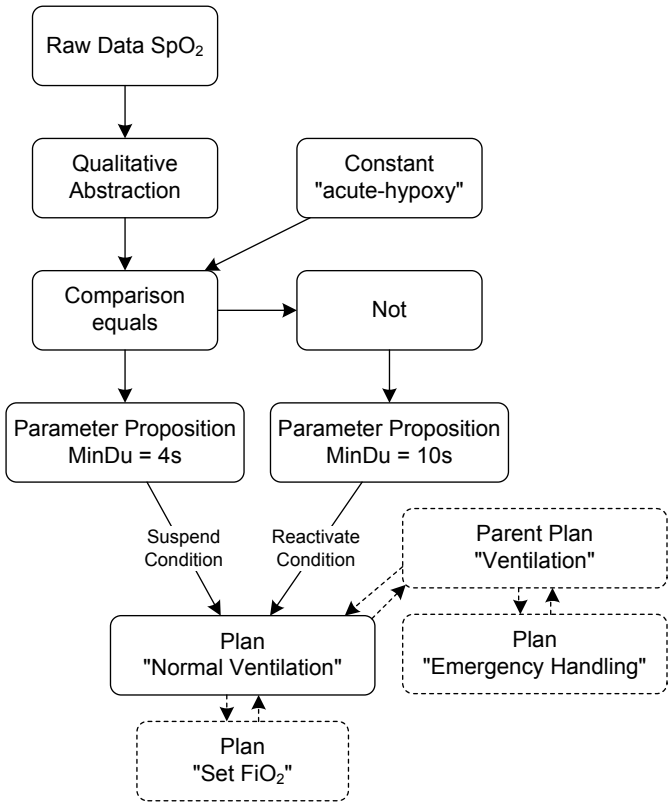


**Figure 4.7:** Sample module graph. A raw data module for the parameter *SpO2* is connected to a qualitative abstraction module. The resulting qualitative value is compared against a qualitative constant. This comparison module is connected to a parameter proposition module used for the suspend condition of the normal ventilation plan module. Another parameter proposition module is connected to the re-activate condition, which uses the negated output of the comparison module as input. The three other plan modules framed with a dashed line are shown to illustrate the context of this example but are not described here.

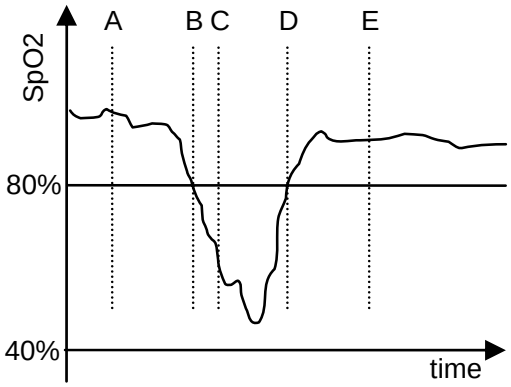


**Figure 4.8:** Sample input data. The graph shows the measurements of saturation of oxygen in blood ($SpO_2$). The horizontal line shows the threshold between the qualitative region of acute hypoxy (below 80%) and the other qualitative regions not relevant in our example. The dotted lines mark relevant time points described in the text.

*4 sec.* The second parameter-proposition contains *SpO2-qualitative not-equal acute hypoxy* and *minimum duration = 10 sec.*

Figure 4.7 on page 60 shows the relevant part of the module graph that is generated by the Asbru Compiler based on this specification.

Let us now look at a typical series of events during abstracting and monitoring the input using the described modules. Figure 4.8 above shows an excerpt from monitored saturation of oxygen in blood ($SpO_2$).

Time point A represents one of many time steps during which the plan *normal ventilation* is activated and no changes are required. The value of *SpO2* is above the threshold (80%). Therefore, the comparison module does not create new output, after outputting false once at program start.

At time point B, the $SpO_2$ value falls below the threshold. Therefore, the comparison module outputs true. The directly connected parameter-proposition module at the left detects a positive flank in its input channel and sets an alarm to *current-time + 4s.*

At time point C (i. e. 4 seconds after B), the alarm set at time point B triggers and since the qualitative abstraction of $SpO_2$ did not change (i. e. no negative flank occurred), the parameter proposition module reports a found episode to the plan module. This means that the suspend condition of the plan gets fulfilled and the plan changes its state to *suspended.*

At time point D, the first parameter-proposition detects a negative flank ($SpO_2$ is no more in the range of acute hypoxy) and changes its output to not-fulfilled. In this case, this input has no consequence for the state of the plan module. The second parameter proposition module detects a positive flank, as the comparison module outputs false now, which is inverted by the

*not* module. Consequently, the parameter proposition module sets an alarm to *current-time + 10s*.

At time point E (i. e. 10 seconds after D), the previously set alarm for the second parameter proposition module triggers and since there has not been negative flank for this module until then, this parameter-proposition reports a found episode to the plan module. Therefore, the re-activate condition of the plan evaluates to true and the plan resumes (i. e. it changes its state back to *activated*).

# Chapter 5

# Algorithms for Monitoring Temporal Patterns

Monitoring patient data requires algorithms to match the real world data with the temporal patterns defined in the Asbru plan library. These algorithms must continually process the input and find matching patterns immediately. Additionally, they should require as little state information as possible. In this chapter I describe adequate algorithms for the most important temporal features of the Asbru language.

## 5.1 Parameter Propositions

As explained in chapter 3, the basic Asbru element to build temporal patterns is the *parameter proposition*. The heart of a parameter proposition is its time annotation.

Table 5.1 lists the intensively used abbreviations for the elements of a time annotation. The values for shifts or durations may be undefined, Table 5.2 shows the defaults for undefined constraints.

| | |
|---|---|
| RP | Reference Point |
| ESS | Earliest Starting Shift |
| LSS | Latest Starting Shift |
| EFS | Earliest Finishing Shift |
| LFS | Latest Finishing Shift |
| MinDu | Minimum Duration |
| MaxDu | Maximum Duration |

**Table 5.1:** Elements of a time annotation and their abbreviations.

| value | default |
|---|---|
| ESS | $-\infty$ |
| LSS | $\infty$ |
| EFS | $-\infty$ |
| LFS | $\infty$ |
| MinDu | 0 |
| MaxDu | $\infty$ |

**Table 5.2:** Defaults for unspecified parts of a time annotation, from [59].

### 5.1.1 Verification of Time Annotations

To be able to create a simple and correct algorithm for monitoring parameter propositions, it is important to detect any illegal input before the start of monitoring. *Illegal* here means a time annotation that cannot match any interval. An example for an illegal time annotation is $ESS > LSS$. If the earliest starting shift is greater than the latest starting shift, no positive flank can match this constraint.

Additionally, we can identify *abnormal* time annotations, which are just sub-optimally formalised constraints. For instance if $EFS - LSS > MinDu$, any matching interval must be longer than $EFS - LSS$ and $MinDu$ will not be a constraint. For the purpose of creating a simple algorithm, abnormal time annotations are not relevant, since the aim is to have as few special cases as possible. Additionally, abnormal time annotations can still match intervals, so we do not want to exclude them from monitoring.

Duftschmid et al. describe ways to verify clinical guidelines [20,21]. In [19] Duftschmid also describes a number of implicit constraints on time annotations, which are also listed in the Asbru Reference Manual [60]. Seyfang recognises that there is a distinction between illegal time annotations and those that are still valid but only defined in a non-optimal way. He differentiates between illegal and abnormal time annotations and lists the following rules for time annotations to be *legal* [59]:

$$ESS \leq LSS \tag{5.1}$$
$$EFS \leq LFS \tag{5.2}$$
$$MinDu \leq MaxDu \tag{5.3}$$
$$EFS - LSS \leq MaxDu \tag{5.4}$$
$$MinDu \leq LFS - ESS \tag{5.5}$$

At this point I continued their work and found that this list is not exhaustive. The list of rules for abnormal time annotations contains among others the following:

$$\begin{aligned} ESS &\leq EFS \\ LSS &\leq LFS \end{aligned}$$

Combining the given rules with (5.1) and (5.2) from above, we can write the restrictions in a slightly different form:

$$\begin{aligned} ESS &\leq LSS \leq LFS \\ ESS &\leq EFS \leq LFS \end{aligned}$$

The implicit rule $ESS \leq LFS$ is now more visible. Examining this restriction shows that a time annotation in violation of this rule is illegal. If the earliest starting shift is equal to or greater than the latest finishing shift, the maximum duration for this interval will be zero or negative. $ESS > LFS$ as well as $MaxDu < 0$ are illegal because an interval cannot end before it started. In our implementation, the duration of even the shortest possible time span of one micro step is considered to be greater than zero. Therefore, even the smallest interval is longer than zero time steps. A time annotation of $MaxDu = 0$ would not match any interval, rendering such a time annotation illegal. Two additional rules must be added:

$$ESS < LFS \tag{5.6}$$

$$0 < MaxDu \tag{5.7}$$

Figure 5.1 shows the extended list of rules for legal time annotations.

$$\begin{aligned} ESS &\leq LSS \\ EFS &\leq LFS \\ ESS &< LFS \\ MinDu &\leq MaxDu \\ 0 &< MaxDu \\ EFS - LSS &\leq MaxDu \\ MinDu &\leq LFS - ESS \end{aligned}$$

**Figure 5.1:** Extended list of rules for the verification of time annotations.

Although an interval cannot be negative, a definition of $MinDu < 0$ only results in an abnormal time annotation. But since a negative duration would confuse our algorithm, we set $MinDu = 0$ for any $MinDu < 0$.

### 5.1.2 Types of Parameter Propositions

We will now focus on finding appropriate algorithms for monitoring parameter propositions, but first we must distinguish three basic types of parameter propositions. The three types are different in their reference point.

- single fixed reference point
- repeated reference point
- the moving reference point *now*

To simplify the modules used for monitoring, the input is reduced to a single Asbru Boolean. Boolean *true* means that the parameter description and the context are met, Boolean *false* or *undefined*[1] means that they are not met. For this abstraction, other modules will be used as input to the parameter proposition modules.

With this simplification we can just concentrate on the *positive flanks (PF)* and the *negative flanks (NF)* (see page 24).

### 5.1.3 Monitoring with a Fixed Reference Point

The algorithm for monitoring parameter propositions are based on those proposed by Seyfang [59]. Seyfang gives a detailed analysis for each. Therefore, I only summarise the existing work here and then concentrate on the work done to actually implement the algorithms in code.

For the fixed reference point it is easier to work on fixed times instead of shifts. This is possible since we need to know the reference point before we can start monitoring anyway.

**Earliest starting time (EST):** The absolute time point $RP + ESS$, or $-\infty$ if ESS is not specified.

**Latest starting time (LST):** The absolute time point $RP + LSS$, or $\infty$ if LSS is not specified.

**Earliest starting time (EFT):** The absolute time point $RP + EFS$, or $-\infty$ if EFS is not specified.

**Latest starting time (LFT):** The absolute time point $RP + LFS$, or $\infty$ if LFS is not specified.

The interval for matching positive flanks ([EST, LST]) and the interval for matching negative flanks ([EFT, LFT]) are now known in absolute time. During the monitoring process we can now easily observe the passing of

[1] In Asbru a *Boolean* can actually have three values: *true*, *false*, and *undefined* (or *unknown*). We interpret a change from *true* to both *false* and *undefined* as a negative flank.

the interval for the positive flank and the passing of the interval for the negative flank. The third interval constraining a fitting interval is relative to the positive flank: [PF + MinDu, PF + MaxDu]. So for each positive flank occurring during [EST, LST] we have to re-calculate the time window during which a negative flank must occur.

Of course not all constraining shifts will be defined all the time. This has the following consequences for the monitoring process.

**EST undefined.** We do not have to wait for EST, the starting interval begins before the start of the monitoring process.

**LST undefined.** Any PF after EST will be accepted, only LFT may end monitoring.

**EFT undefined.** Any NF before LFT is suitable, the finishing interval begins before the start of the monitoring process.

**LFT undefined.** The latest time for NF is only constrained by $MaxDu + PF$. Since this constraint is renewed at each PF, there is no direct constraint for the latest NF, however it is indirectly constrained by the latest possible PF.

**MinDu undefined.** There is no minimum duration, but if EFT has not yet passed, NF is constrained by EFT.

**MaxDu undefined.** The latest time for NF is given by LFT alone.

**LST and LFT undefined.** There is always a chance that a matching interval can be found in the future. Monitoring does not end early.

**LFT and MaxDu undefined.** Once a suitable PF is found, the interval is a fitting interval already at EFT or $PF + MinDu$, because NF cannot come too late.

To monitor the passing of the time windows, *alarms* will be used. For the purpose of this description we look at alarms and the positive and negative flanks as uniform events.

Figure 5.2 shows the state chart for monitoring presented in [59]. As the author notes the state chart is not a completely formal representation because it mixes events (e. g. *PF arrived*) with conditions (e. g. $MaxDu = LFT = \infty$) for the state transitions. Therefore, great care must be taken when transforming the state machine into code.

As the author also notes, implementing Figure 5.2 node by node and arc by arc as finite state machine seems redundant. Instead he presents pseudo code for an algorithm based on four state variables. The three state variables *SI (starting interval)*, *DI (during interval)*, and *FI (finishing interval)* can

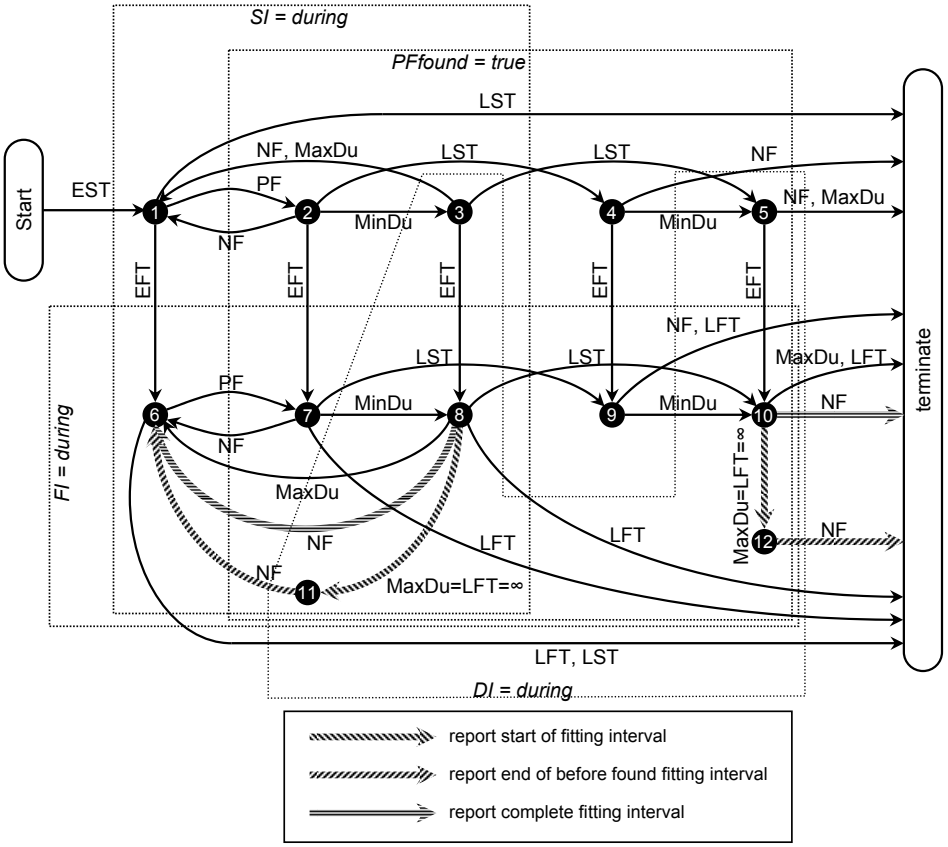


**Figure 5.2:** State chart of monitoring a parameter proposition with a fixed reference point. Wide, stripped arcs stand for the action "output description of fitting interval" performed during these state transitions. This action is split in cases where the interval is found to fit before it ends. From [59].

have the values *before*, *during*, and *past*. The state variable *PFfound* can be *true* or *false*.

Each state is represented as an area with dotted borders in the state chart (Figure 5.2). The areas are labelled with the state variable's short name. The starting interval (SI) is the interval during which PF must occur to start a fitting interval. The finishing interval (FI) is the interval during which NF must occur. The during interval (DI) is defined by MinDu and MaxDu with PF. It starts with MinDu being over and ends with MaxDu, LFT, or NF. PFfound tells if a fitting PF has been found.

### Implementation of the Proposed Algorithm

Based on the suggested pseudo code in [59] I have developed the algorithm presented as the function processEvent on page 71.

The algorithm is based on the four state variables described above. Before the start of monitoring the constraining intervals are checked according to the rules in section 5.1.1. As we internally define the start of monitoring as time 0 (zero), the conditions $LST >= 0$ and $LFT > 0$ must be fulfilled, too. If the time annotation is illegal (i. e. it cannot match any interval), monitoring is not started for this parameter proposition and *end of monitoring* will be reported.

At the start of monitoring the state variables are initialised. If EST is undefined ($-\infty$) or is before the start of monitoring ($< 0$), any positive flank before or equal to LST[2] will match the starting interval (SI). Therefore, the starting interval is set to *during* in that case, otherwise to *before*. The same is done for EFT and the finishing interval (FI). The during interval (DI) is set to *before*, PFfound to *false*. For EST, LST, EFT, and LFT alarms are registered appropriately.

The monitoring itself happens in the processEvent function. According to each *event* the state variables are changed and resulting changes of the output are reported. Nevertheless there are several cases when more than one state transition in the original state chart (Figure 5.2) is required in one step. This is because some transitions are based on input (*events, e. g. PF*) whereas others are based on conditions (i. e. $MaxDu = LFT = \infty$). Some of them can be based on both, e. g. MinDu can mean the event *minimum duration expired* or the condition $MinDu = 0$.

**Undefined constraints.** As we have just seen with the example of MinDu, undefined constraints of the three intervals may require the algorithm to explicitly check the condition at each state that has an outgoing transition labelled with the respective constraint. Let me analyse the constraints in this regard. Since an undefined maximum never implies an action[3] (the interval does not expire), only the minimum constraints are relevant. That leaves EST, EFT, and MinDu. Undefined values for EST and EFT are already taken care of at the start of monitoring. Therefore, we only need to explicitly check for undefined MinDu and the explicitly given condition $MaxDu = LFT = \infty$.

So when must $MinDu = 0$ be checked? Looking at the state chart we can see that this condition must be checked in states 2, 4, 7, and 9. Into these states we come through PF, EFT, and LST. Thus, if we check $MinDu = 0$ directly after PF and EFT, we can even ignore LST because there is no way to move through LST on $2 \rightarrow 4$ or $7 \rightarrow 9$. We will always move $2 \rightarrow 3$ or $7 \rightarrow 8$ first. Conclusion: The algorithm must check an undefined MinDu after PF and EFT.

$MaxDu = LFT = \infty$ must be checked in states 8 and 10, to which we

[2] LST must be $>= 0$, otherwise the time annotation is illegal.

[3] The exception to the rule, $MaxDu = LFT = \infty$, is explicitly given in the state chart.

come through EFT, MinDu (states 8, 10), and LST ($8 \rightarrow 10$). As above, there is no need to check after LST because checking for $MaxDu = LFT = \infty$ in state 8 already moves the state $8 \rightarrow 11$. Conclusion: The algorithm must check $MaxDu = LFT = \infty$ after EFT and MinDu.

**Result.** In my implementation I have solved the processing of additional conditions by recursively calling processEvent in those cases. The main part is given as pseudo code of processEvent on page 71. A description for each event with references to the state transitions in the chart is given below.

**EST** Earliest starting time passed. SI (starting interval) is set to *during*, corresponding to transition $Start \rightarrow 1$ in the state chart.

**LST** Latest starting time passed. If we have a valid positive flank right now ($PFfound = true$), set SI (starting interval) to *past* ($2 \rightarrow 4,\ 3 \rightarrow 5,\ 7 \rightarrow 9,\ 8 \rightarrow 10$), otherwise terminate ($1, 6 \rightarrow terminate$).

**EFT** Earliest finishing time passed. Set FI (finishing interval) to *during* ($1 \rightarrow 6,\ 2 \rightarrow 7,\ 3 \rightarrow 8,\ 4 \rightarrow 9,\ 5 \rightarrow 10$). Then, if a positive flank was found, check for $MinDu = 0$ and implicitly $MaxDu = LFT = \infty$ (recursively call processEvent($MinDuEx$), as explained above).

**LFT** Latest finishing time passed, terminate ($6, 7, 8, 9, 10 \rightarrow terminate$).

**PF** Positive flank found. If the positive flank is within the starting interval (SI = *during*), set PFfound to *true*, set DI (during interval) to *before*, and set the duration alarms for MinDu and MaxDu ($1 \rightarrow 2,\ 6 \rightarrow 7$). Also check for $MinDu = 0$ as explained above.

**MinDuEx** Minimum duration expired, set DI (during interval) to *during* ($2 \rightarrow 3,\ 4 \rightarrow 5,\ 7 \rightarrow 8,\ 9 \rightarrow 10$). If FI (finishing interval) is *during*, check for $MaxDu = LFT = \infty$. If so, report *start of interval* (NF is not yet known; $8 \rightarrow 11,\ 10 \rightarrow 12$).

**NF** Negative flank found. If we have a valid positive flank, FI (finishing interval) is *during* and DI (during interval) is *during*, then if $MaxDu = LFT = \infty$ report *end of before-found interval* (*start of interval* was reported before; $11 \rightarrow 6,\ 12 \rightarrow terminate$), otherwise report *complete interval* ($8 \rightarrow 6,\ 10 \rightarrow terminate$). Additionally, if we are still in the starting interval (SI), set PFfound to *false* ($2 \rightarrow 1;\ 7, 8, 11 \rightarrow 6$), otherwise add *end of monitoring* to the output and terminate ($4, 5, 9, 10, 12 \rightarrow terminate$).

**MaxDuEx** Maximum duration expired. Set DI (during interval) to *past*. If we are still in the starting interval (SI), set PFfound to *false* ($3 \rightarrow 1,\ 8 \rightarrow 6$), otherwise terminate reporting *end of monitoring* ($5, 10 \rightarrow terminate$).

**Function** `processEvent(`*event*`)`. Pseudo code for monitoring of a parameter proposition with fixed reference point.

```
Data: SI starting-, FI finishing-, DI during-interval, PFfound positive flank
function processEvent(event)
    switch event do
        case EST:
            SI ← during
        case LST:
            if PFfound then
                SI ← past
            else
                result ← end of monitoring
        case EFT:
            FI ← during
            if PFfound and ( MinDu = 0 or DI = during ) then
                result ← processEvent(MinDuEx)
        case LFT:
            result ← end of monitoring
        case PF:
            if SI = during then
                PFfound ← now
                DI ← before
                set alarms for MinDu, MaxDu
                if MinDu = 0 then result ← processEvent(MinDuEx)
        case MinDuEx:
            DI ← during
            if FI = during and MaxDu = LFT = ∞ then
                result ← start of interval
        case NF:
            if PFfound and FI = during and DI = during then
                if MaxDu = LFT = ∞ then
                    result ← end of before-found interval
                else
                    result ← complete interval
            if SI = during then
                PFfound ← false
            else
                result ← result ∪ end of monitoring
        case MaxDuEx:
            DI ← past
            if SI = during then
                PFfound ← false
            else
                result ← end of monitoring
    return result
```

The algorithm for monitoring parameter propositions with a fixed reference point is implemented in the class FixedParameterPropositionModule. The algorithm has been tested to give correct and timely answer for all inputs. For testing and verification see chapter 6.

### 5.1.4 Monitoring with Repeated Reference Points

Monitoring a parameter proposition with repeated reference points can be simply seen as iterative monitoring with a fixed reference point and calculating the union of the individual episodes as output, i. e. eliminating duplicated episodes found with more than on reference point. Duplicate episodes are episodes that have the same positive flank.

Each individual module for a specific reference point must run until *end of monitoring* is reported for this specific instance. Because the distance of reference points can be shorter than the time until end of monitoring, an arbitrary number of modules must be created and run in parallel. The parameters defining the maximum number of needed instances are only known at runtime, therefore, implementing a dedicated module for each reference point requires modifications of the module graph at runtime.

Another approach would be to create a single module to have multiple instances of the state machine to monitor each reference point. The algorithm to monitor each of those would still be the same as described in the previous subsection.

In the current implementation, the first approach is taken, i. e. for each new reference point a FixedParameterPropositionModule is created and added to the module graph. As soon as the module reports *end of monitoring*, the module is removed from the graph. This is implemented in the RepeatedParameterPropositionModule.

### 5.1.5 Monitoring with Reference Point *Now*

In contrast to a parameter proposition with a fixed or repeated reference point, monitoring with reference point *now* is different in several aspects:

1. We cannot define any constraint based on fixed time points because the intervals are constantly moving.

2. There is no end of monitoring because of the first statement.[4]

3. Again, because the constraints are moving, a fitting interval is usually only fitting for a constrained (shorter or longer) time frame, after which it is no longer matching the constraints. The start of this time frame is called *start of validity*, the end is called *end of validity*.

[4] The monitoring is only stopped for this parameter proposition if the information is not needed any more.

4. Any valid shift must be zero or negative, i. e. point to the past, because we can at no point look into the future. So a time annotation with positive shifts would never result in a fitting interval.[5]

By now the constraints for positive and negative flanks were fixed in time. We imagined the time and therefore the flanks as moving. To understand monitoring with reference point *now*, we should consider the flanks as fixed in time (after they occurred) and then calculate the window of during which the interval will be a fitting interval, based on the flanks and the constraints. If such a time window is found, *start* and *end of validity* is output at start and end of this window. What does not change from the case with a fixed reference point is the processing of the minimum and maximum duration as those have already depended on the positive flank. A more detailed analysis is given below.

### Calculating the Effective Duration Constraints

The constraints that a time annotation imposes on a fitting interval, consist of two parts, the duration of the interval and the location on the time axis. Let me consider the duration first. Obviously, it is constrained by the minimum and maximum duration (*MinDu* and *MaxDu*). As we have seen before, the minimum duration is also set by the latest starting shift and the earliest finishing shift. The maximum duration is also defined by the earliest starting shift and the latest finishing shift. The effective minimum and maximum durations are therefore defined as follows [59]:

$$\begin{aligned} EffMinDu &= max(EFS - LSS,\ MinDu) \\ EffMaxDu &= min(LFS - ESS,\ MaxDu) \end{aligned}$$

So as soon as a positive flank is found, a negative flank must now occur between $PF + EffMinDu$ and $PF + EffMaxDu$. Alarms for the duration constraints are set at the positive flank.

### Calculating Start and End of Validity

For the validity of an interval, i. e. the time window during which an interval is a fitting interval, we look at the flanks again. At PF we can say that for this interval, the window of validity will start at PF plus the absolute value of the (negative) latest starting shift, $PF + |LSS|$, or $PF - LSS$. If it sounds odd to a reader that the *start* of validity should be constrained by the *latest* starting shift, remember that the *time* and thus also the latest starting time is moving forward. For example, a second after the start of

[5] When compiling the plan library, positive numbers in the shifts of a time annotation with reference *now* are mapped to negative numbers for user convenience.

validity, $now - |LSS|$ will be after PF, and PF will actually be earlier than the latest starting time now.

If the minimum duration is defined, the start of validity is also constrained by $PF + MinDu$ because we cannot know that the interval is a fitting interval before MinDu has passed. Accordingly we must define the start of validity based on the negative flank, $NF - LFS$.

$$\begin{aligned} SV_{PF} &= PF - LSS \\ SV_{NF} &= NF - LFS \\ SV_{MinDu} &= PF + MinDu \end{aligned}$$

The end of validity is calculated from PF and ESS, and NF and EFS.

$$\begin{aligned} EV_{PF} &= PF - ESS \\ EV_{NF} &= NF - EFS \end{aligned}$$

To calculate the actual time window of validity for a found interval we must consider all these five constraints. For a more detailed analysis see [59].

$$\begin{aligned} SV &= max(SV_{PF}, SV_{NF}, SV_{MinDu}) \\ EV &= min(EV_{PF}, EV_{NF}) \end{aligned}$$

The pseudo code for calculating the start and end of validity for real values is given in algorithm 5.2. The methods are used to calculate the validity as soon as a fitting interval is found. For start of validity the default value is *now* because this is the earliest time we can know that the fitting interval is valid anyway. When there are no constraints on the earliest starting or finishing time, there is no end of validity, therefore, the default value is positive infinite ($\infty$).

Figure 5.3 on page 76 shows the state chart for monitoring with reference point *now*. Note that the automaton must be able to take several steps in immediate succession. Equally to the state chart for a fixed reference point, some arcs are labelled with events while others are labelled with queries.

### Implementation of the Proposed Algorithm

The algorithm proposed by Seyfang lacks explicit output of *end of validity*. In the case of a constrained maximum duration, i. e. $MaxDu \neq \infty \vee LFS \neq \infty$, the fitting interval is only known at NF, and therefore, SV and EV are both known at the report of the complete interval. For the case of an unconstrained maximum duration, i. e. $MaxDu = LFS = \infty$, the situation

**Algorithm 5.2**: Pseudo code for calculating the start and end of validity of a found interval for monitoring of a parameter proposition with reference point *now*.

```
Result: Start of validity for found interval.
function getSV(now, PF, NF)
    SV_PF, SV_NF, SV_MinDu ← now
    if LSS defined then
        SV_PF ← PF − LSS
    if NF known then
        if LFS defined then
            SV_NF ← NF − LFS
        else
            SV_NF ← NF
    SV_MinDu ← PF + MinDu
    return max(SV_PF, SV_NF, SV_MinDu)

Result: End of validity for found interval.
function getEV(now, PF, NF)
    EV_PF, EV_NF ← ∞
    if ESS defined then
        EV_PF ← PF − ESS
    if NF known and EFS defined then
        EV_NF ← NF − EFS
    return max(EV_PF, EV_NF)
```

is different. At $PF + MinDu$ the start of a fitting interval must be reported. It is then quite simple to come up with an example time annotation for which end of validity will be before the negative flank. In such a case, *end of validity* must be reported appropriately first and *end of before-found fitting-interval* later as soon as NF occurs.

My implementation fulfils this requirement by implementing the algorithm based on the two state variables *PFfound* (PF was found) and *MinDuPassed* (minimum duration passed after PF), and the additional variable *EVbeforeNF* (end of validity was reported before the negative flank). At start of monitoring, these state variables are set to *false*.

**Output** When monitoring with a fixed reference point, the output of the algorithm was always valid at the time it was reported. Now, monitoring with reference point *now*, we will often find information that is known now, but must be output in the future. For example, if the latest starting shift is $-5$ hours, and there is no constraint on the maximum duration, the start of validity will be $now + 5\ hours$ earliest and should only be reported to other modules in the monitoring process at that time. My framework implementation makes this quite easy by allowing output with a future valid time. The Execution Manager keeps the output for us until the valid time occurs

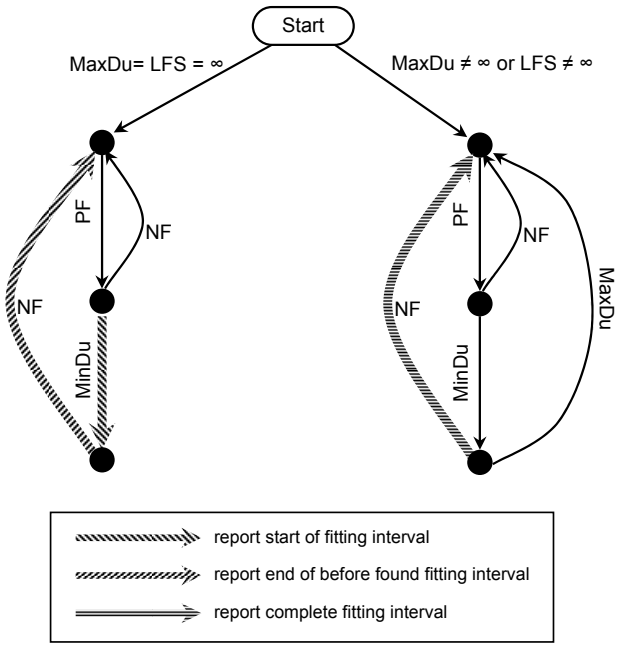


**Figure 5.3:** State chart of monitoring a parameter proposition with reference point *now*. Wide, stripped arcs stand for the action "output description of fitting interval" performed during these state transitions. This action is split in cases where the interval is found to fit before it ends. From [59].

and only then reports the information to other modules. In the pseudo code this is illustrated by e. g., *report start of validity* ***at SV***, which means to report the start of validity, but with a valid time of the start of the validity time window.

**Result** The algorithm for monitoring a parameter proposition with reference point *now* is given as pseudo code of the function processEvent on page 77. A description for each event is given below.

**PF** Positive flank found. Set PFfound to now, MinDuPassed and EVbeforeNF to false. Set alarms for MinDu (at $PF + MinDu$) and MaxDu (at $PF + MaxDu$). If the effective minimum duration is zero, recursively call processEvent(*MinDuEx*) to either set MinDuPassed to true or, if there is no constraint on the maximum duration, report the found interval.

**NF** Negative flank found. If there is a currently valid positive flank (i. e. not expired), check if MinDuPassed is true, set PFfound to false and cancel any duration alarms.

**Function** `processEvent(`*event*`)`. Pseudo code for monitoring of a parameter proposition with reference point *now*.

**Data**: MinDuPassed if MinDu has passed, EVbeforeNF if EV has happened before NF, PFfound positive flank found

```
function processEvent(event)
    switch event do
        case PF:
            PFfound ← now
            MinDuPassed, EVbeforeNF ← false
            set alarms for MinDu, MaxDu
            if EffMinDu = 0 then
                result ← processEvent(MinDuEx)
        case NF:
            if PFfound then
                if MinDuPassed then
                    if MaxDu = LFS = ∞ then
                        result ← end of before-found interval
                        if not EVbeforeNF then
                            EV ← getEV(now, PFfound, now)
                            if EV defined then
                                result ← result ∪ end of validity at EV
                                cancel alarm for EV
                    else
                        SV ← getSV(now, PFfound, now)
                        result ← start of validity at SV
                        EV ← getEV(now, PFfound, now)
                        if EV defined then
                            result ← result ∪ end of validity at EV
                PFfound ← false
                cancel alarms for MinDu, MaxDu
        case MinDuEx:
            MinDuPassed ← true
            if MaxDu = LFS = ∞ then
                SV ← getSV(now, PFfound, undefined)
                result ← start of validity at SV
                EV ← getEV(now, PFfound, undefined)
                if EV defined then
                    set alarm for EV before NF at EV
        case EVBeforeNF:
            EVbeforeNF ← true
            result ← end of validity
        case MaxDuEx:
            PFfound ← false
    return result
```

If MinDuPassed is true, then, if there is no maximum duration constraint report *end of before-found interval* and *end of validity* if it has not been reported yet. If there is a maximum duration constraint, report the complete interval now.

**MinDuEx** Minimum duration expired. Set MinDuPassed to true. If there is no maximum duration constraint, report start of validity. If end of validity based on the positive flank ($EV_{PF}$) is defined, i. e. ESS is defined, register an alarm for *EV before NF* at EV.

**EVBeforeNF** End of validity occurred before the negative flank. Set EVbeforeNF to true and report end of validity now.

**MaxDuEx** Maximum duration expired. Set PFfound to false.

The algorithm for monitoring a parameter proposition with reference point *now* is implemented in the class MovingParameterPropositionModule (see the class overview in chapter 4). The class EpisodeDataPoint is used to report information in any of the three parameter proposition modules as well as all other modules presented in this chapter. Its properties and the requirements for any module using this data point class are described in detail in section 4.2.3.

## 5.2 Temporal Constraints

Asbru temporal constraints are based on Allen's thirteen temporal relations as first described in [2]. Temporal relations, e. g. *before*, *overlaps*, or *equal*, are used in the natural language to describe information about events and intervals in time. Allen formalised these relations in an interval-based temporal logic to create algorithms for computational reasoning about intervals [2, 3]. Kaiser et al. [38] have analysed possible temporal relations between different plan states and plan state transitions in Asbru plans to gain knowledge and provide means for the verification of guidelines.

However, Asbru temporal constraints themselves are not used to gain knowledge about temporal intervals or events. Instead they are used to gain knowledge about a patient's health condition by matching patient data with patterns defined as temporal relations in the plan library. The task is to find all instances of a given temporal relation of episodes from two streams. This requires a distinctive set of algorithms.

Table 5.3 lists the seven temporal constraints defined in Asbru. Each of the other six of the thirteen possible relations is an inverse of a given one and can be achieved by exchanging the arguments.

| Relation | Definition |
|---|---|
| A before B | $NF_A < PF_B$ |
| A meets B | $NF_A = PF_B$ |
| A overlaps B | $PF_A < PF_B \wedge PF_B < NF_A < NF_B$ |
| A starts B | $PF_A = PF_B \wedge NF_A < NF_B$ |
| A during B | $PF_A > PF_B \wedge NF_A < NF_B$ |
| A finishes B | $PF_A > PF_B \wedge NF_A = NF_B$ |
| A equal B | $PF_A = PF_B \wedge NF_A = NF_B$ |

**Table 5.3:** Allen's temporal relations translated into Asbru temporal constraints. The constraints are defined based on the positive and negative flanks of the intervals.

**Output.** The output of a temporal constraint is an episode for each matching pair of input episodes (from inputs A and B). The positive flank for the output episode is the earlier positive flank ($PF_{out} = min(PF_A, PF_B)$), the negative flank is the later negative flank of the two inputs ($NF_{out} = max(NF_A, NF_B)$). In other words, the output episodes' flanks enclose the input episodes. The interval of validity must be the intersection of the intervals of validity of the pair of input episodes.

**Input Ordering.** Seyfang describes algorithms to monitor temporal constraints at near constant computational costs per episode, i. e. O($n$) where $n$ is the number of all processed input episodes [59]. He achieves this by simplifying the problem assuming that the temporal patterns arrive totally ordered by the positive flank at the monitoring module. In practice this simplification can be achieved by delaying information (based on the maximum data delay for a given input) so that data from both inputs is processed in order. Since the current framework implementation does not support individual data delay yet, another approach is needed. A different issue is that some temporal patterns will always generate output that cannot be processed in order without an *infinite* delay of the processing.

A simple example of such a temporal pattern is the *before* constraint. Any episode of input A will be *before* any later episode of input B. Therefore, for any later episode of input B, the module needs to output the combinations of all earlier episodes from input A with the given episode from input B. The start (positive flank) of the output episodes will not be ordered but instead goes back to the first episode of input A for each episode of input B (compare Figure 6.1 on page 98). Because of this, it would be impossible to monitor the output of two *before* constraints for *equality* for example.

For the given reasons I have created algorithms that accept unordered input. This is achieved at the expense of higher computational costs. It must be said that the aim was not to find the fastest possible algorithm (from a runtime perspective), but a correct algorithm that could act as a reference

| Operator | Definition with epsilon |
|---|---|
| $a = b$ | $\|a - b\| \leq epsilon$ |
| $a > b$ | $a > b - epsilon$ |
| $a \geq b$ | $a \geq b - epsilon$ |
| $a < b$ | $a < b + epsilon$ |
| $a \leq b$ | $a \leq b + epsilon$ |

**Table 5.4:** Definition of (in)equality operators with epsilon.

implementation for future optimisation work. A sensible enhancement will be to implement the algorithms proposed by Seyfang for the cases where this is possible, and use mine in the other cases.

### 5.2.1 Temporal Constraints and Epsilon

The reference point of an Asbru parameter-proposition may also be defined with an *epsilon-region*, which denotes a region around the time point given as the reference. This means, that the actual time point can lie somewhere between the reference minus *epsilon* and the reference plus *epsilon*.

Asbru provides *epsilon* for temporal constraints, too. In this context, epsilon is useful under some circumstances to describe "fuzzy" matches within defined bounds. If data is received from different sources, the data might not be exactly synchronised, so when monitoring for equality the actual positive and negative flanks of two intervals might by off by some tenth of a second, compared to an interval length of some seconds or minutes, for example. If these two intervals should match as equal, epsilon provides means to achieve this.

Some of the cases where epsilon seems useful could be described in different ways by combining several temporal constraints (e. g., *A before B* ***or*** *A meets B*) with appropriate parameter propositions to limit how much an episode A could be before B to match the *before* constraint. This technique might help to solve some but not many of the use cases for epsilon and is rather inefficient.

In the pseudo code given in the next subsections, *epsilon* is not explicitly included for readability. Instead, the redefinition of the operators given in Table 5.4 is assumed. For example, if the pseudo code reads $NF(a) < NF(b)$, the comparison will be $NF(a) < NF(b)+epsilon$ for epsilons greater than zero.

### 5.2.2 Algorithm Design

The algorithms I have created for monitoring temporal constraints are based on a single design for all types of constraints. The common implementation

is done in the Java class TemporalConstraintModule. The algorithm uses two sorted sets, i. e. one for each input, and a list of already output episodes. The sorted input sets provide a sorted representation of the inputs currently relevant for processing and additionally means to add and remove inputs from the set with $O(\ln n)$ behaviour. The latter is needed to efficiently remove revoked inputs. The sorted view is needed to find matching inputs with a runtime of $O(1)$ best case to $O(n)$ worst case, where $n$ is the number of currently valid inputs in the opposite input of the currently processed one. This is currently done using a standard Java TreeMap.

The output list is used for two purposes. First it records input episodes that are known to match, but are not yet known complete (*negative flank* unknown), second it is used to keep track of output episodes that must be revoked if an input is revoked later. Nevertheless there are certain monitoring modules that output episodes that will never be revoked, for example monitoring parameter propositions with a fixed reference point. Optimisation is possible here because waiting for the revocation of an episode (i. e. *end of validity*) is not necessary. Because of this, each episode carries the information if it is revocable, in other words if there can be an *end of validity* of this episode. If none of the inputs of a pair of episodes constituting an output episode is revocable, the output episode can be removed from the list as soon as the interval is known-complete, i. e. both positive flank and negative flank are known.

**Basic task.** The task for monitoring at each time step is:

- For each input process *start of validity* and *end of before-found interval* of any episodes. For each input episode, match the input against existing, valid inputs that are potential fits. Eliminating episodes that are no potential fits is a key aspect in a fast algorithm here. Found matches are added to the output list.
- Process *end of validity* of any input episodes. For each finished episode, revoke all already output episodes.
- If both input modules reported *end of monitoring*, create an output episode representing end of monitoring for this module.

Item two and three above are the same for all temporal constraints and only need one common implementation. The difference between the different temporal relations is only in the part actually matching the episodes. This is implemented for each temporal constraint in its own subclass of TemporalConstraintModule as method processEvent.

As said above, eliminating episodes that are no potential fit is important. When matching a given episode, the list of inputs of the other input channel is searched. We want to find the most probable matches first in the list

and also have an abort condition that allows us to ignore all other inputs as not matching. The list of episodes can be sorted by either positive flank or negative flank, and the list can be searched forward or backward. Which of the flanks is most significant for matching depends on the given temporal relation we want to monitor.

For example looking at the *A before B* relation, the negative flank of an episode $a$ must occur before the positive flank of an episode $b$. Therefore, the inputs of A should be sorted by the negative flank and the inputs of B by the positive flank. When analysing an instance of B, the most probable match for *A before B* is the oldest instance of A. Therefore, the instances of A will be sorted in ascending order. An analysis for the best order of the inputs for each temporal constraint is given below, together with the definition and the algorithm for each.

### 5.2.3 The Temporal Constraint *A before B*

Two intervals A and B match the constraint *A before B* if and only if interval A ends before interval B starts. Based on the flanks of the intervals, the constraint *A before B* is defined as:

$$NF_A < PF_B$$

**Input Ordering.** As explained earlier, the input sets are kept as sorted set to enable fast lookups of matching episode pairs. For the before-relation, the best approach is to sort episodes of input A by the negative flank ascending and episodes of input B by the positive flank descending.

**Algorithm Description.** The pseudo code for input matching is given as function processEvent on page 83 and is implemented in EpisodeBeforeModule.

The processEvent method receives two arguments, first the type of information this event represents (one of $PF_A$, $PF_B$, $NF_A$, $NF_B$), second the episode data as one element out of an EpisodeDataPoint. Note that if both positive flank and negative flank are known at the initial report of an episode, the method will still be called for $PF$ and $NF$ separately, so we need not think about this special case. The method must return new output as list of EpisodeDataPoints and additionally store or update output in the *output list*. The list below describes how the algorithm reacts on each kind of input.

$PF_A$ At the positive flank of an input A we cannot know if the episodes can match an episode B. The event could be ignored. Nevertheless the episode must be stored to keep track of it in case that the episode becomes invalid (*end of validity*) before the negative flank is reported. Therefore, the episode is stored in the input set A.

**Function** `processEvent(`*event, episode*`)`. Pseudo code for monitoring of the temporal constraint ***A before B***.

**Data**: $\mathsf{Set}_A$ sorted set of episodes from input A, ordered by NF ascending, *undefined* last. $\mathsf{Set}_B$ sorted set of episodes from input B, ordered by PF descending. OutputList list of episodes that have been output.

```
function processEvent(event, episode)
    switch event do
        case PF_A:
            add episode to Set_A
        case PF_B:
            add episode to Set_B
            {Find all episodes of Set_A that ended before episode}
            foreach a ∈ Set_A do
                if NF(a) undefined then
                    break
                if NF(a) < PF(episode) then
                    add pair (a, episode) to OutputList
                    add start of validity of (a, episode) to result
                else
                    break
        case NF_A:
            if not episode ∈ Set_A then
                break
            update episode in Set_A
            {Find all episodes of Set_B that started after episode}
            foreach b ∈ Set_B do
                if NF(episode) < PF(b) then
                    add pair (episode, b) to OutputList
                    add start of validity of (episode, b) to result
                else
                    break
        case NF_B:
            if episode ∈ Set_B then
                update episode in Set_B
            {Find all output episodes that contain this episode and report
            end of before-found interval for each}
            foreach output ∈ OutputList do
                if episode part of output then
                    add end of before-found interval of output to result
                    if not Revocable(output) or Revoked(output) then
                        remove output from OutputList
    return result
```

$PF_B$ Store the episode in the input set B. Then find all episodes of input set A that ended before this episode. We can abort the search if either an episode A with an unknown negative flank (sorted last) is found or the negative flank of an episode A is greater than or equal to the positive flank of the input episode.

$NF_A$ If the episode cannot be found in the input set, end of validity has already occurred and this event will be skipped. It provides no new information relevant for the output of this module.

If the episode is still valid, update the episode in the input set A and find all instances in input set B that started after this episode.

$NF_B$ Update the episode in the input set B. Then find all output episodes that contain this input episode and report *end of before-found interval* for them. Revoked episodes and episodes based on inputs that were not revocable can now be removed from the output list.

### 5.2.4 The Temporal Constraint *A meets B*

Two intervals A and B match the constraint *A meets B* if and only if interval A ends exactly when interval B starts. Based on the flanks of the intervals, the constraint *A meets B* is defined as:

$$NF_A = PF_B$$

**Input Ordering.** To find matches for this constraint, we need to compare the negative flank of episodes from input A with the positive flank of episodes from input B. Therefore, the input set A will be sorted by the negative flank, whereas the input set B will be sorted by the positive flank.

In contrast to the *before* relation we cannot say whether the most probable matches will be the earliest or the latest episode, but instead they will be those that are close to each other. The best way to find potential matches would be to use binary search in the sorted set to find matches. Nevertheless I have made the assumption that the delay between a positive flank and the start of validity is shorter than the time from start to end of validity. As long as this assumption is true, we can find most matches in the most recently found episodes. Both input sets are therefore sorted in descending order.

**Epsilon.** When *epsilon* is used with this temporal constraints, there can be unexpected results because only $NF_A$ and $PF_B$ are checked, which is sufficient if epsilon is not considered. With epsilon, a relation matching *overlaps* or *before* can also match *meets*. This is the desired behaviour. However intervals in any relation potentially match *meets*, if epsilon is higher than the interval length. Common sense must say that epsilon should be very

low compared to the expected interval lengths and so this is not a problem, but it should be mentioned.

**Algorithm Description.** The pseudo code for input matching is given as function processEvent on page 85 and is implemented in EpisodeMeetsModule.

---

**Function** `processEvent(`*event, episode*`)`. Pseudo code for monitoring of the temporal constraint ***A meets B***.

---

**Data**: $\mathsf{Set}_A$ sorted set of episodes from input A, ordered by NF descending, *undefined* last. $\mathsf{Set}_B$ sorted set of episodes from input B, ordered by PF descending. OutputList list of episodes that have been output.

```
function processEvent(event, episode)
    switch event do
        case PF_A:
            add episode to Set_A
        case PF_B:
            add episode to Set_B
            {Find all episodes of Set_A so that a meets episode}
            foreach a ∈ Set_A do
                if NF(a) undefined then break
                if NF(a) = PF(episode) then
                    add pair (a, episode) to OutputList
                    add start of validity of (a, episode) to result
                else if not NF(a) ≥ PF(episode) then
                    break
        case NF_A:
            if not episode ∈ Set_A then break
            update episode in Set_A
            {Find all episodes of Set_B so that episode meets b}
            foreach b ∈ Set_B do
                if NF(episode) = PF(b) then
                    add pair (episode, b) to OutputList
                    add start of validity of (episode, b) to result
                else if not NF(episode) ≤ PF(b) then
                    break
        case NF_B:
            {Same as NF_B for the before constraint above}
    return result
```

---

The list below describes how the algorithm reacts on each kind of input.

$PF_A$ Same as for *A before B*. The episode is stored in the input set A.

$PF_B$ Store the episode in the input set B. Find all episodes of input set A that end exactly at $PF_B$.

$NF_A$ If the episode cannot be found in the input set, end of validity has already occurred and this event will be skipped. If the episode is still valid, update the episode in the input set A and find all instances in input set B that start exactly at $NF_A$.

$NF_B$ Same as for *A before B*. Update the episode in the input set B. Then find all output episodes that contain this input episode and report *end of before-found interval* for them.

### 5.2.5 The Temporal Constraint *A overlaps B*

Two intervals A and B match the constraint *A overlaps B* if and only if interval A starts before the start of interval B, and the end of interval A lies between the start and the end of interval B. Based on the flanks of the intervals, the constraint *A overlaps B* is defined as:

$$PF_A < PF_B \ \wedge \ PF_B < NF_A < NF_B$$

**Input Ordering.** The *overlaps* constraint is similar to the *meets* constraint. The negative flank of input A and the positive flank of input B are the significant properties to compare. The input set A will be sorted by the negative flank and the input set B will be sorted by the positive flank again. The sort order will be descending for both.

**Algorithm Description.** Different from above is that we must also check $PF_A$ and more importantly $NF_B$ to be sure that two intervals match. $NF_B$ will usually not be known at $NF_A$ for two matching intervals, but it is sufficient to know that $NF_B$ is later than $NF_A$, the exact time of $NF_B$ is irrelevant. Of course, the algorithm must take care of this special case because of the principle that all facts must be reported as soon as they *can* be known. The implementation is simplified by determining that input B will be processed before input A. So when we find $NF_A$, and an episode in the input set B does not have the negative flank defined, we know that it will occur later than now. This is implemented in the superclass TemporalConstraintModule.

The pseudo code for input matching is given as function processEvent on page 87 and is implemented in EpisodeOverlapsModule.

The list below describes how the algorithm reacts on each kind of input.

$PF_A$ Same as for *A before B*. The episode is stored in the input set A.

$PF_B$ Store the episode in the input set B. Find all episodes of input set A that fulfil the constraint *A overlaps B*. If the negative flank of episode B is unknown, it must occur later than any known negative flank of episodes from set A.

**Function** `processEvent(`*event, episode*`)`. Pseudo code for monitoring of the temporal constraint ***A overlaps B***.

**Data**: $\mathsf{Set}_A$ sorted set of episodes from input A, ordered by NF descending, *undefined* last. $\mathsf{Set}_B$ sorted set of episodes from input B, ordered by PF descending. OutputList list of episodes that have been output.

```
function processEvent(event, episode)
    switch event do
        case PF_A:
            add episode to Set_A
        case PF_B:
            add episode to Set_B
            {Find all episodes of Set_A so that a overlaps episode}
            foreach a ∈ Set_A do
                if NF(a) undefined then break
                if NF(a) > PF(episode) then
                    if PF(a) < PF(episode) and (NF(a) < NF(episode)
                    or NF(episode) undefined) then
                        add pair (a, episode) to OutputList
                        add start of validity of (a, episode) to result
                else
                    break
        case NF_A:
            if not episode ∈ Set_A then break
            update episode in Set_A
            {Find all episodes of Set_B so that episode overlaps b}
            foreach b ∈ Set_B do
                if PF(episode) < PF(b) then
                    if NF(episode) > PF(b) and (NF(episode) < NF(b)
                    or NF(b) undefined) then
                        add pair (episode, b) to OutputList
                        add start of validity of (episode, b) to result
                else
                    break
        case NF_B:
            {Same as NF_B for the before constraint above}
    return result
```

$NF_A$ If the episode cannot be found in the input set, end of validity has already occurred and this event will be skipped. If the episode is still valid, update the episode in the input set A and find all instances in input set B that fulfil the constraint *A overlaps B*. If the negative flank of any episode B is unknown, it must occur later than the given negative flank of episode A.

$NF_B$ Same as for *A before B*. Update the episode in the input set B. Then find all output episodes that contain this input episode and report *end of before-found interval* for them.

### 5.2.6 The Temporal Constraint *A starts B*

Two intervals A and B match the constraint *A starts B* if and only if interval A and B start together, and interval A ends before interval B ends. Based on the flanks of the intervals, the constraint *A starts B* is defined as:

$$PF_A = PF_B \ \wedge \ NF_A < NF_B$$

**Input Ordering.** For the *starts* constraint, $PF_A$ and $PF_B$ are the most significant flanks that must be compared. The negative flanks must also be checked, and as for the *overlaps* constraint the algorithm must take care of the case when $NF_A$ is known and $NF_B$ is not yet known but known to occur later.

The input sets A and B will be sorted by the positive flanks of the episodes. Both will be sorted in descending order to find the latest episodes first.

**Algorithm Description.** The pseudo code for input matching is given as function processEvent on page 89 and is implemented in EpisodeStartsModule.

The list below describes how the algorithm reacts on each kind of input.

$PF_A$ Same as for *A before B*. The episode is stored in the input set A.

$PF_B$ Store the episode in the input set B. Find all episodes of input set A that fulfil the constraint *A starts B*. If the negative flank of episode B is unknown, it must occur later than any known negative flank of episodes from set A.

$NF_A$ Same as for *A overlaps B*. If the episode cannot be found in the input set, end of validity has already occurred and this event will be skipped. If the episode is still valid, update the episode in the input set A and find all instances in input set B that fulfil the constraint *A starts B*. If the negative flank of any episode B is unknown, it must occur later than the given negative flank of episode A.

**Function** `processEvent(`*event, episode*`)`. Pseudo code for monitoring of the temporal constraint ***A starts B***.

**Data**: $\mathsf{Set}_A$ sorted set of episodes from input A, ordered by PF descending.
$\mathsf{Set}_B$ sorted set of episodes from input B, ordered by PF descending.
OutputList list of episodes that have been output.

```
function processEvent(event, episode)
    switch event do
        case PF_A:
            add episode to Set_A
        case PF_B:
            add episode to Set_B
            {Find all episodes of Set_A so that a starts episode}
            foreach a ∈ Set_A do
                if NF(a) undefined then continue
                if PF(a) = PF(episode) then
                    if NF(a) < NF(episode) or NF(episode) undefined
                    then
                        add pair (a, episode) to OutputList
                        add start of validity of (a, episode) to result
                else if not PF(a) ≥ PF(episode) then
                    break
        case NF_A:
            if not episode ∈ Set_A then break
            update episode in Set_A
            {Find all episodes of Set_B so that episode starts b}
            foreach b ∈ Set_B do
                if PF(episode) = PF(b) then
                    if NF(episode) < NF(b) or NF(b) undefined then
                        add pair (episode, b) to OutputList
                        add start of validity of (episode, b) to result
                else if not PF(episode) ≤ PF(b) then
                    break
        case NF_B:
            {Same as NF_B for the before constraint above}
    return result
```

$NF_B$ Same as for *A before B*. Update the episode in the input set B. Then find all output episodes that contain this input episode and report *end of before-found interval* for them.

### 5.2.7 The Temporal Constraint *A during B*

Two intervals A and B match the constraint *A during B* if and only if interval A starts after the start of interval B and ends before the end of interval B. Based on the flanks of the intervals, the constraint *A during B* is defined as:

$$PF_A > PF_B \ \wedge \ NF_A < NF_B$$

**Input Ordering.** For the *during* constraint it is not immediately obvious which flanks should be used for the input ordering. Using the positive flank of episode A for sorting has the advantage that we will find our abort condition faster when searching through the list. When processing episodes from input A, sorting episodes from input B by the negative flank, with unknown negative flanks first, will sort the potential matches first. Unknown negative flanks of episodes from input B will again mean that they will occur later than any known negative flank of input A.

**Algorithm Description.** The pseudo code for input matching is given as function processEvent on page 91 and is implemented in EpisodeDuringModule.

The list below describes how the algorithm reacts on each kind of input.

$PF_A$ Same as for *A before B*. The episode is stored in the input set A.

$PF_B$ Store the episode in the input set B. Find all episodes of input set A that fulfil the constraint *A during B*. If the negative flank of episode B is unknown, it must occur later than any known negative flank of episodes from set A.

$NF_A$ Same as for *A overlaps B*. If the episode cannot be found in the input set, end of validity has already occurred and this event will be skipped. If the episode is still valid, update the episode in the input set A and find all instances in input set B that fulfil the constraint *A during B*. If the negative flank of any episode B is unknown, it must occur later than the given negative flank of episode A.

$NF_B$ Same as for *A before B*. Update the episode in the input set B. Then find all output episodes that contain this input episode and report *end of before-found interval* for them.

**Function** `processEvent(`*event, episode*`)`. Pseudo code for monitoring of the temporal constraint ***A during B***.

**Data**: $\mathsf{Set}_A$ sorted set of episodes from input A, ordered by PF descending. $\mathsf{Set}_B$ sorted set of episodes from input B, ordered by NF descending, *undefined* first. OutputList list of episodes that have been output.

```
function processEvent(event, episode)
    switch event do
        case PF_A:
            add episode to Set_A
        case PF_B:
            add episode to Set_B
            {Find all episodes of Set_A so that a during episode}
            foreach a ∈ Set_A do
                if NF(a) undefined then continue
                if PF(a) > PF(episode) then
                    if NF(a) < NF(episode) or NF(episode) undefined
                    then
                        add pair (a, episode) to OutputList
                        add start of validity of (a, episode) to result
                else
                    break
        case NF_A:
            if not episode ∈ Set_A then break
            update episode in Set_A
            {Find all episodes of Set_B so that episode during b}
            foreach b ∈ Set_B do
                if not NF(b) undefined and not PF(episode) < NF(b)
                then break
                if PF(episode) > PF(b) and (NF(episode) < NF(b) or
                NF(b) undefined) then
                    add pair (episode, b) to OutputList
                    add start of validity of (episode, b) to result
        case NF_B:
            {Same as NF_B for the before constraint above}
    return result
```

### 5.2.8 The Temporal Constraint *A finishes B*

Two intervals A and B match the constraint *A finishes B* if and only if interval A starts after the start of interval B, and interval A and B finish together. Based on the flanks of the intervals, the constraint *A finishes B* is defined as:

$$PF_A > PF_B \ \wedge \ NF_A = NF_B$$

**Input Ordering.** For the *finishes* constraint, $NF_A$ and $NF_B$ are the most significant flanks that must be compared. Here it is not necessary to compare episodes with unknown negative flanks. Both, input set A and B will be sorted by the negative flank in descending order with unknown negative flanks last.

**Algorithm Description.** The pseudo code for input matching is given as function processEvent on page 93 and is implemented in EpisodeFinishesModule.

The list below describes how the algorithm reacts on each kind of input.

$PF_A$ Same as for *A before B*. The episode is stored in the input set A.

$PF_B$ Same as $PF_A$. We need to know the negative flanks of episode A and B, therefore, the episode is just stored in input set B.

$NF_A$ If the episode cannot be found in the input set, end of validity has already occurred and this event will be skipped.
If the episode is still valid, update the episode in the input set A and find all instances in input set B that fulfil the constraint *A finishes B*.

$NF_B$ If the episode cannot be found in the input set, end of validity has already occurred and this event will be skipped.
If the episode is still valid, update the episode in the input set B and find all instances in input set A that fulfil the constraint *A finishes B*.

**Function** `processEvent(`*event, episode*`)`. Pseudo code for monitoring of the temporal constraint ***A finishes B***.

**Data**: $\mathsf{Set}_A$ sorted set of episodes from input A, ordered by NF descending, *undefined* last. $\mathsf{Set}_B$ sorted set of episodes from input B, ordered by NF descending, *undefined* last. OutputList list of episodes that have been output.

```
function processEvent(event, episode)
    switch event do
        case PF_A:
            add episode to Set_A
        case PF_B:
            add episode to Set_B
        case NF_A:
            if not episode ∈ Set_A then break
            update episode in Set_A
            {Find all episodes of Set_B so that episode finishes b}
            foreach b ∈ Set_B do
                if NF(b) undefined then break
                if NF(episode) = NF(b) then
                    if PF(episode) > PF(b) then
                        add pair (episode, b) to OutputList
                        add start of validity of (episode, b) to result
                else if not NF(episode) ≤ NF(b) then
                    break
        case NF_B:
            if not episode ∈ Set_B then break
            update episode in Set_B
            {Find all episodes of Set_A so that a finishes episode}
            foreach a ∈ Set_A do
                if NF(a) undefined then break
                if NF(a) = NF(episode) then
                    if PF(a) > PF(episode) then
                        add pair (a, episode) to OutputList
                        add start of validity of (a, episode) to result
                else if not NF(a) ≥ NF(episode) then
                    break
    return result
```

### 5.2.9 The Temporal Constraint *A equal B*

Two intervals A and B match the constraint *A finishes B* if and only if interval A and interval B start and end together. Based on the flanks of the intervals, the constraint *A equal B* is defined as:

$$PF_A = PF_B \wedge NF_A = NF_B$$

**Input Ordering.** For the *equal* constraint, the positive and negative flank of both intervals must match exactly. Because we need to know the complete intervals anyway, both input sets will be sorted by the negative flank in descending order, unknown negative flanks last.

**Algorithm Description.** The pseudo code for input matching is given as function processEvent on page 95 and is implemented in EpisodeEqualModule.

The list below describes how the algorithm reacts on each kind of input.

$PF_A$ Same as for *A before B*. The episode is stored in the input set A.

$PF_B$ Same as $PF_A$. The episode is stored in the input set B.

$NF_A$ Same as for *A finishes B*. If the episode cannot be found in the input set, end of validity has already occurred and this event will be skipped. If the episode is still valid, update the episode in the input set A and find all instances in input set B that fulfil the constraint *A equal B*.

$NF_B$ If the episode cannot be found in the input set, end of validity has already occurred and this event will be skipped.
If the episode is still valid, update the episode in the input set B and find all instances in input set A that fulfil the constraint *A equal B*.

**Function** `processEvent(`*event, episode*`)`. Pseudo code for monitoring of the temporal constraint ***A equal B***.

**Data**: $\mathsf{Set}_A$ sorted set of episodes from input A, ordered by NF descending, *undefined* last. $\mathsf{Set}_B$ sorted set of episodes from input B, ordered by NF descending, *undefined* last. OutputList list of episodes that have been output.

```
function processEvent(event, episode)
    switch event do
        case PF_A:
            add episode to Set_A
        case PF_B:
            add episode to Set_B
        case NF_A:
            if not episode ∈ Set_A then break
            update episode in Set_A
            {Find all episodes of Set_B so that episode equal b}
            foreach b ∈ Set_B do
                if NF(b) undefined then break
                if NF(episode) = NF(b) then
                    if PF(episode) = PF(b) then
                        add pair (episode, b) to OutputList
                        add start of validity of (episode, b) to result
                else if not NF(episode) ≤ NF(b) then
                    break
        case NF_B:
            if not episode ∈ Set_B then break
            update episode in Set_B
            {Find all episodes of Set_A so that a equal episode}
            foreach a ∈ Set_A do
                if NF(a) undefined then break
                if NF(a) = NF(episode) then
                    if PF(a) = PF(episode) then
                        add pair (a, episode) to OutputList
                        add start of validity of (a, episode) to result
                else if not NF(a) ≥ NF(episode) then
                    break
    return result
```

# Chapter 6

# Evaluation and Conclusion

In this chapter, I focus on the testing and verification done on the implementation developed as part of this thesis. A discussion of the current state of the Asbru Interpreter, including desirable future improvements, follows. A summary and conclusion completes this chapter as well as the thesis.

## 6.1 Testing Individual Components

There are basically three phases of software testing: unit testing, integration testing, and system testing. Unit testing is used to validate a particular module of source code. The test cases should be as independent as possible from each other. Integration testing groups modules in larger aggregates and then applies tests. Integration testing aims at verifying requirements on subsystems of an implementation. System testing is conducted on a complete, integrated system.

Testing a software system is generally far from trivial. Testing an execution engine for a guideline modeling language, especially one as complex as Asbru, is an even more demanding task. The set of possible inputs to the system is increased by the fact that the "program" is actually defined by the plan library. Therefore, it is even more important to identify subsystems that actually can be tested against a semantic specification.

Several groups of tests ensure that the Asbru Interpreter exactly meets the elaborate semantics of the Asbru Language. This includes for example testing of the temporal abstraction components or the verification of the correct plan propagation semantics (see section 6.2).

This section describes some aspects of the testing done on the subsystems of the Asbru Interpreter, especially the integration tests of the monitoring modules for parameter propositions, temporal constraints, and constraint combinations. Section 6.2 describes the conducted system tests. For several important components, unit tests have been created. These class-level tests are not further described in this thesis.

| Reference | ESS | LSS | EFS | LFS | MinDu | MaxDu |
|---|---|---|---|---|---|---|
| now | | | 0 min | | | |
| now | | | -10 min | -5 min | 2 min | 10 min |
| now | -10 min | -1 min | -1 min | | | |
| now | | | 0 min | | 1 min | |
| self | | | | 20 min | 1 min | 2 min |
| self | | | 5 min | | 10 min | |
| self | -10 min | 10 min | 0 min | 10 min | 10 min | 15 min |

**Table 6.1:** Examples of time annotations used for the tests of the parameter proposition module.

### 6.1.1 Parameter Proposition

A desirable target for testing the parameter proposition would be to not only have complete path coverage of the source code of the module, but to also cover all possible states of the state machine. Remember that the parameter proposition module for a fixed reference point basically uses three three-state variables and one two-state variable. There are eight different events triggering state transitions, although not all events are possible in all states as long as the outside system (mainly the Execution Manager) works correctly. The code paths for the events additionally depend on the shifts of the time annotation, more specifically, whether specific shifts are defined or not.

It would be complex to devise a method to create all those test cases required to achieve the code coverage described above. Another option would be to use a *brute-force* testing approach to generate all possible qualitatively different time annotations as well as input sets. An optimistic, educated guess is that this would require at least $10^7$ test runs.

Due to constraints on the available time, I had to chose a less complete approach. Assuming that the state chart in Figure 5.2 on 68 is complete and correct (which is not yet proven), I tried to show in section 5.1.3 that my algorithm is a correct and complete implementation of this state chart.

Then I created a set of test cases that execute the most commonly used as well as several unusual time annotations to show that at least many cases work as expected. Table 6.1 shows a selection of these test cases. A file with time-stamped Boolean values is used as input. The results are compared to a manually calculated results table.

### 6.1.2 Temporal Constraints

When testing temporal constraint modules, an important factor to pay attention to is that the inputs may be received out-of-order. Therefore, it is

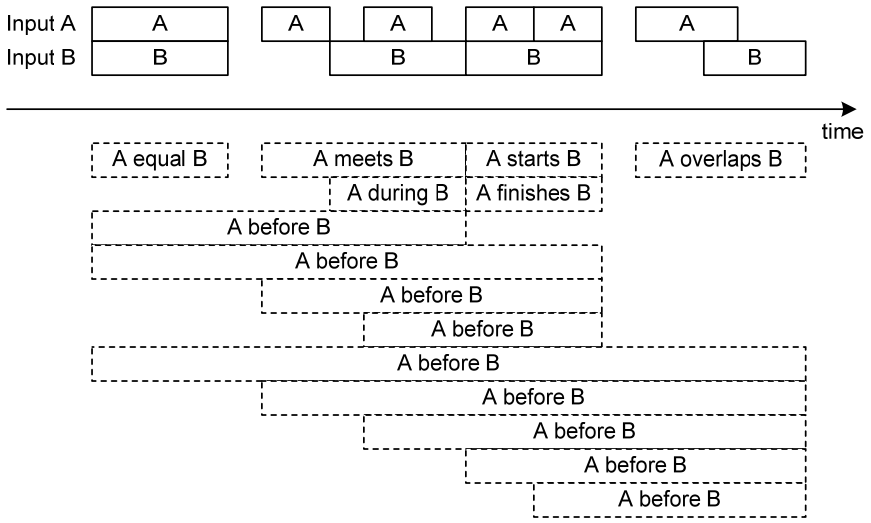


**Figure 6.1:** Inputs and found episodes for temporal constraints tests. Above the x-axis the input episodes from input A and B are shown. Below, the enclosing intervals of the temporal relations found by the respective temporal constraint modules are shown. Note the huge and overlapping episodes found for the *before* relation.

necessary to test the modules with valid and invalid input from parameter propositions with different data-delay properties.

Figure 6.1 schematically shows the inputs for the temporal constraints tests. For each input channel, six different parameter propositions with different time annotations are created. Each of the six parameter propositions for input A is monitored for the seven temporal relations with each of the six parameter propositions for input B, resulting in 36 different timing combinations and 252 created constraint combination modules.

The output of each module is checked for correctness. The input pattern given in the figure is repeated twice. Additionally, the input sequence is reversed to test for negative matches.

### 6.1.3 Constraint Combination

For the trivial case of two or more parameter propositions with the time annotation *now* (i. e. RP=now, EFS=0), the result of a constraint combination of these parameter propositions is equal to the result of the Boolean combination of the inputs. It is therefore easy to test the modules for a large number of inputs without the need to calculate the expected results in advance.

Figure 6.2 shows the module graph for testing the constraint combination *and*. Two Boolean inputs are monitored using the parameter proposition

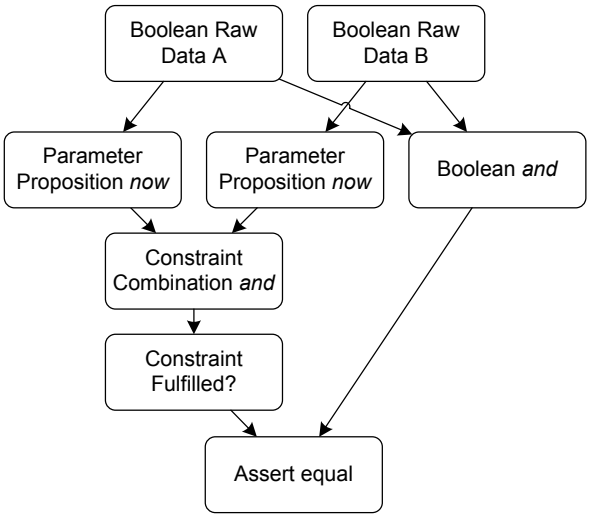


**Figure 6.2:** Module graph for testing constraint combinations. Two Boolean inputs are monitored using the parameter proposition *now*. The output of the constraint combination *and* is compared with the regular Boolean *and* of the Boolean inputs. The results must match. The graph shows the setup for the *and* combination, tests for *or* and *xor* work the same way.

*now*. The output of the constraint combination *and* is compared to the regular Boolean *and* of the Boolean inputs. The results must match. This is repeated for all constraint combinations. A similar setup can be used to test the *constraint-not* module.

In addition to these tests, I created test cases resembling the ones described for the temporal constraint modules. These tests check that the flanks of the issued episodes are also correct.

## 6.2 The Asbru Interpreter: System Tests

During my work on the implementation of the framework for the Interpreter, members of the Protocure project started to implement the compiler components as well as the modules required for plan execution. As soon as the first version of the interpreter began to take shape, a framework was created to evaluate the plan execution semantics of the execution unit.

The test framework creates 2000+ prototypical plan configurations based on plan library templates. These tests evaluate the correct plan propagation for all temporal orderings of subplans as well as propagation specifications. The tests ensure that the Asbru Interpreter exactly meets the elaborate semantics of the Asbru language.

The integration tests as well as the system tests have proven to be invaluable tools for regression testing during the ongoing development of the

software. Without automated tests, a project of such a complexity would be infeasible.

### 6.2.1 Performance Tests

Asbru is well suited for guidelines and protocols in high-frequency domains. A design goal for the interpreter was to achieve an efficient implementation that would be usable in domains such as intensive care. Data in this domain is usually recorded at 1 Hz or 200 Hz.

Performance tests show that the interpreter can process input from several channels and moderately complex abstractions thereof at a rate of more than 1 kHz on a standard notebook PC.

We have found that the single most relevant bottleneck in the system currently is the execution trace log, which accumulates to several megabytes of XML data in a few seconds of processing data at a frequency of 1 kHz. The trace log can be disabled, which improves the performance by at least a factor of three. The complete trace log (which includes every single data point created by all the modules) is usually used in low-frequency domains to evaluate compliance with a guideline. For high-frequency domains, it is only used for selected sample scenarios, to validate the implementation. In these cases, performance is not critical and the data volume is small. A further improvement would be the implementation of a selective filter to extract the interesting parts of the execution trace at runtime, before the trace is converted into a stream of XML data. Currently, the filtering is performed by XSLT scripts in post-processing the log.

## 6.3 Future Work

By now, the Asbru Interpreter supports a subset of the available language features (Asbru Light). Although the feature set is not complete, a general usage is certainly possible. Further work is required to implement the remainder of the Asbru language elements.

As mentioned in section 2.2.4 in the context of the Knowledge-Based Temporal-Abstraction method, clinical data often arrives out of temporal order. For example, analysis of a blood sample in a clinical laboratory might by available only several hours after the blood sample was taken. On the other hand, sensor data is available immediately. There are two different temporal dimensions involved here – the time when the blood sample was taken, and the time when the data is available in the system. To allow the representation of these different temporal dimensions, Snodgrass et al. have proposed to store a *transaction time* as well as a *valid time* for each datum [71].

Implementing temporal reasoning in an efficient way is hard when data does not arrive temporally ordered (compare section 2.2.4). Therefore, Sey-

fang [59] proposes to use a concept called *data delay* to delay the processing of individual input channels of the system. The data delay can be individually specified for each input channel, so that the delay for sensor measurements can be very low, whereas the delay for an analysis in a clinical laboratory can be set to a reasonable amount of time.

This is currently not implemented in the system. The Execution Manager does not distinguish between the two temporal dimensions and is not able to handle different data delays. For the clinical guidelines that have been evaluated so far, this is not a problem, even less for batch processing of patient data to analyse a guideline. Nevertheless, in a clinical setting with a mixture of high-frequency and low-frequency data, the implementation would need revision. Due to the architecture of the interpreter and the forethought in designing the components, it will be possible to achieve this goal by rewriting only a small portion of the execution code.

## 6.4 Conclusion

Guideline-based care plays an increasingly important part in clinical environments. Automated application of clinical guidelines and protocols requires the abstraction of raw-data into higher-level medical concepts. The most powerful tools available require an immense amount of computing-resources because they create all possible abstractions without any information, which will ever be used or queried.

Asbru's methodology is unique in that it allows temporal abstraction and reasoning to be integrated with guideline-based plan execution. In Asbru, temporal abstraction can be implemented in a resource-efficient way, because the required temporal abstractions are limited to the concepts used for the specific guideline or protocol (i. e. by a restriction of the problem space).

In this thesis I analysed the requirements for implementing the modular framework for execution of Asbru guidelines proposed by Seyfang [59] and the Protocure team[1]. Based on the gathered knowledge I defined exact and sound semantics for the modules and the interactions between the various components. I then described my implementation, which was illustrated as part of the Asbru Interpreter with an example from the field of ventilation of neonates.

New algorithms were required to implement Asbru's temporal abstraction based on [59]. The algorithms for monitoring parameter propositions have a constant-time cost of $O(1)$ for each new input (or $O(n)$ for the complete input set over the total runtime). This fact together with the efficient implementation of the Execution Manager are the foundation of the outstanding performance of the system and its eligibility for data-driven plan execution in high-frequency domains.

[1] Protocure project homepage: `http://www.protocure.org`. Accessed Feb 2, 2006.

# Bibliography


[1] Wolfgang Aigner and Silvia Miksch. Supporting protocol-based care in medicine via multiple coordinated views. In J. C. Roberts, P. Rodgers, and N. Boukhelifa, editors, *Proceedings of the Second International Conference on Coordinated and Multiple Views in Exploratory Visualization (CMV 2004)*, pages 118–129. IEEE Computer Society Press, 2004.

[2] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[3] James F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, July 1984.

[4] Juan Carlos Augusto. Temporal reasoning for decision support in medicine. *Artificial Intelligence in Medicine*, 33(1):1–24, 2005.

[5] Michael Balser, Christoph Duelli, and Wolfgang Reif. Formal semantics of Asbru – an overview. In H. Ehrig, B. Kraemer, and A. Ertas, editors, *Proceedings of the 6th World Conference on Integrated Design and Process Technology (IDPT-02)*, pages 1–8, Pasadena, CA, 2002. Society for Design and Process Science.

[6] Riccardo Bellazzi, Cristiana Larizza, Paolo Magni, and Roberto Bellazzi. Quality assessment of hemodialysis services through temporal data mining. In Michel Dojat, Elpida Keravnou, and Pedro Barahona, editors, *Artificial Intelligence in Medicine. Proceedings of the 9th Conference on Artificial Intelligence in Medicine in Europe (AIME 2003)*, pages 11–20, Protaras, Cyprus, 2003. Springer-Verlag.

[7] Riccardo Bellazzi, Yuval Shahar, et al. Intelligent data analysis in medicine and pharmacology (panel summary). Held during during AMIA 1999 Annual Symposium. Washington, DC, November 1999.

[8] David Boaz and Yuval Shahar. A framework for distributed mediation of temporal-abstraction queries to clinical databases. *Artificial Intelligence in Medicine*, 34(1):3–24, 2005.

[9] Tibor Bosse. An interpreter for clinical guidelines in asbru. Master's thesis, Vrije Universiteit Amsterdam, 2001.

[10] Aziz A. Boxwala, Mor Peleg, Samson W. Tu, Omolola Ogunyemi, Qing T. Zeng, Dongwen Wang, Vimla L. Patel, Robert A. Greenes, , and Edward H. Shortliffe. GLIF3: a representation format for shareable computer-interpretable clinical practice guidelines. *Journal of Biomedical Informatics*, 37(3):147–161, June 2004.

[11] Shubha Chakravarty and Yuval Shahar. A constraint-based specification of periodic patterns in time-oriented data. In *6th International Workshop on Temporal Representation and Reasoning (TIME '99)*, pages 29–40, Orlando, FL, 1999. IEEE Computer Society Press.

[12] Sylvie Charbonnier. On-line extraction of temporal episodes from icu high-frequency data: A visual support for signal interpretation. *Computer Methods and Programs in Biomedicine*, 78:115–132, May 2005.

[13] Paolo Ciccarese, Ezio Caffi, Lorenzo Boiocchi, Assaf Halevy, Silvana Quaglini, Anand Kumar, and Mario Stefanelli. The NewGuide project: Guidelines, information sharing and learning from exceptions. In Michel Dojat, Elpida Keravnou, and Pedro Barahona, editors, *Artificial Intelligence in Medicine. Proceedings of the 9th Conference on Artificial Intelligence in Medicine in Europe (AIME 2003)*, pages 163–167, Protaras, Cyprus, 2003. Springer-Verlag.

[14] Paolo Ciccarese, Ezio Caffi, Lorenzo Boiocchi, Silvana Quaglini, and Mario Stefanelli. A guideline management system. In Marius Fieschi et al., editors, *Proceedings of the 11th World Congress on Medical Informatics (Medinfo 2004)*, pages 28–32, San Antonio, TX, 2004. IOS Press.

[15] Carlo Combi, Giancarlo Cucchi, and Francesco Pinciroli. Applying object-oriented technologies in modeling and querying temporally oriented clinical databases dealing with temporal granularity and indeterminacy. *IEEE Transactions on Information Technology in Biomedicine*, 1(2):100–127, June 1997.

[16] Carlo Combi and Yuval Shahar. Temporal reasoning and temporal data maintenance in medicine: Issues and challenges. *Computers in Biology and Medicine*, 27:353–368, September 1997.

[17] Amar K. Das and Mark A. Musen. A temporal query system for protocol-directed decision support. *Methods of Information in Medicine*, 33(4):358–370, 1993.

[18] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

[19] Georg Duftschmid. *Knowledge-Based Verification of Clinical Guidelines by Detection of Anomalies*. PhD thesis, Vienna University of Technology, Vienna, January 1999.

[20] Georg Duftschmid and Silvia Miksch. Knowledge-based verification of clinical guidelines by detection of anomalies. *OEGAI Journal*, 18(2):37–99, 1999.

[21] Georg Duftschmid and Silvia Miksch. Knowledge-based verification of clinical guidelines by detection of anomalies. *Artificial Intelligence in Medicine*, 22(1):23–41, April 2001.

[22] Marilyn J. Field and Kathleen H. Lohr, editors. *Clinical Practice Guidelines: Directions for a New Program*, Institute of Medicine, Washington DC, 1990. National Academy Press.

[23] John Fox and Subrata Das. *Safe and Sound: Artificial Intelligence in Hazardous Applications*. AAAI Press/MIT Press, 2000.

[24] John Fox, Nicky Johns, and Ali Rahmanzadeh. Disseminating medical knowledge: the PROforma approach. *Artificial Intelligence in Medicine*, 14(1–2):157–181, September 1998.

[25] Peter E. Friedland and Yumi Iwasaki. The concept and implementation of skeletal plans. *Journal of Automated Reasoning*, 1(2):161–208, 1985.

[26] Christian Fuchsberger, Jim Hunter, and Paul McCue. Testing Asbru guidelines and protocols for neonatal intensive care. In Silvia Miksch, Jim Hunter, and Elpida T. Keravnou, editors, *Artificial Intelligence in Medicine. Proceedings of the 10th Conference on Artificial Intelligence in Medicine in Europe (AIME 2005)*, pages 101–110, Aberdeen, UK, 2005. Springer-Verlag.

[27] Christian Fuchsberger and Silvia Miksch. Asbru's execution engine: Utilizing guidelines for artificial ventilation of newborn infants. In *Proceedings of the Joint Workshop Intelligent Data Analysis in Medicine and Pharmacology and Knowledge-Based Information Management in Anaesthesia and Intensive Care (IDAMAP and KBIM-AIC 2003), Workshop at the 9th Conference on Artificial Intelligence in Medicine in Europe (AIME 2003)*, pages 99–125, Protaras, Cyprus, 2003.

[28] Mary K. Goldstein, Robert W. Coleman, Samson W. Tu, Ravi D. Shankar, Martin J. O'Connor, Mark A. Musen, Susana B. Martins, Philip W. Lavori, Michael G. Shlipak, Eugene Oddone, Aneel A. Advani, Parisa Gholami, and Brian B. Hoffman. Operationalizing clinical

practice guidelines amidst changing evidence: Athena, an easily modifiable decision-support system for management of hypertension in primary care. *Journal of the American Medical Informatics Association*, 9(6 Suppl 1):11–16, 2002.

[29] Mary K. Goldstein, Robert W. Coleman, Samson W. Tu, Ravi D. Shankar, Martin J. O'Connor, Mark A. Musen, Susana B. Martins, Philip W. Lavori, Michael G. Shlipak, Eugene Oddone, Aneel A. Advani, Parisa Gholami, and Brian B. Hoffman. Translating research into practice: Organizational issues in implementing automated decision support for hypertension in three medical centers. *Journal of the American Medical Informatics Association*, 11(5):368–376, 2004.

[30] Jeremy M. Grimshaw and Ian T. Russell. Effects of clinical guidelines on medical practice: a systematic review of rigorous evaluation. *Lancet*, 342:1317–22, November 1993.

[31] Ira J. Haimowitz and Isaac S. Kohane. Managing temporal worlds for medical trend diagnosis. *Artificial Intelligence in Medicine*, 8(3):299–321, July 1996.

[32] Werner Horn. AI in medicine on its way from knowledge-intensive to data-intensive systems. *Artificial Intelligence in Medicine*, 23(1):5–12, 2001.

[33] George Hripcsak, Peter Ludemann, T. Allan Pryor, Ove B. Wigertz, and Paul D. Clayton. Rationale for the Arden syntax. *Computers in Biomedical Research*, 27(4):291–324, August 1994.

[34] Jim Hunter and Neil McIntosh. Knowledge-based event detection in complex time series data. In *AIMDM '99: Proceedings of the Joint European Conference on Artificial Intelligence in Medicine and Medical Decision Making*, pages 271–280, London, UK, 1999. Springer-Verlag.

[35] Peter Johnson, Samson W. Tu, and Neill Jones. Achieving reuse of computable guideline systems. In Vimla L. Patel et al., editors, *Proceedings of the 10th World Congress on Medical Informatics (Medinfo 2001)*, pages 99–103, London, UK, 2001. IOS Press.

[36] Peter D. Johnson, Samson W. Tu, Nick Booth, Bob Sugden, and Ian N. Purves. Using scenarios in chronic disease management guidelines for primary care. In J. Marc Overhage, editor, *Converging Information, Technology, and Health Care. Proceedings of the AMIA 2000 Annual Symposium*, pages 389–393, Los Angeles, CA, 2000. Hanley and Belfus.

[37] Mary E. Johnston, Karl B. Langton, R. Brian Haynes, and Alix Mathieu. Effects of computer-based clinical decision support systems on clin-

ician performance and patient outcome: a critical appraisal of research. *Annals of Internal Medicine*, 120(2):135–142, January 1994.

[38] Katharina Kaiser and Silvia Miksch. Treating temporal information in plan and process modeling. Technical Report Asgaard-TR-2004-1, Vienna University of Technology, Institute of Software Technology and Interactive Systems, Vienna, February 2004.

[39] Robert Kosara and Silvia Miksch. Metaphors of movement: A visualization and user interface for time-oriented, skeletal plans. *Artificial Intelligence in Medicine*, 22(2):111–131, 2001.

[40] Cristiana Larizza, Riccardo Bellazzi, and Giordano Lanzola. An HTTP-based server for temporal abstractions. In *Proceedings of the Fourth Workshop on Intelligent Data Analysis in Medicine and Pharmacology (IDAMAP-99)*, pages 52–62, Washington, DC, 1999.

[41] Silvia Miksch. Plan management in the medical domain. *AI Communications*, 12(4):209–235, 1999.

[42] Silvia Miksch and Andreas Seyfang. Continual planning with time-oriented, skeletal plans. In Werner Horn, editor, *ECAI 2000, Proceedings of the 14th European Conference on Artificial Intelligence*, pages 511–515, Berlin, Germany, 2000. IOS Press.

[43] Silvia Miksch, Andreas Seyfang, Werner Horn, and Christian Popow. Abstracting steady qualitative descriptions over time from noisy, high-frequency data. In *AIMDM '99: Proceedings of the Joint European Conference on Artificial Intelligence in Medicine and Medical Decision Making*, pages 281–290, London, UK, 1999. Springer-Verlag.

[44] Silvia Miksch, Andreas Seyfang, and Christian Popow. Abstraction and representation of repeated patterns in high-frequency data. In Nada Lavrac, Silvia Miksch, and Branko Kavsek, editors, *Proceedings of the Fifth Workshop on Intelligent Data Analysis in Medicine and Pharmacology (IDAMAP-2000), Workshop Notes of the 14th European Conference on Artificial Intelligence (ECAI-2000)*, pages 32–39, Berlin, Germany, 2000.

[45] Silvia Miksch, Yuval Shahar, and Peter Johnson. Asbru: A task-specific, intention-based, and time-oriented language for representing skeletal plans. In E. Motta et al., editors, *7th Workshop on Knowledge Engineering: Methods and Languages (KEML-97)*, Milton Keynes, UK, 1997.

[46] Mark A. Musen, Samson W. Tu, Amar K. Das, and Yuval Shahar. EON: A component-based approach to automation of protocol-directed

therapy. *Journal of the American Medical Informatics Association*, 3(6):367–388, 1996.

[47] John H. Nguyen, Yuval Shahar, Samson W. Tu, Amar K. Das, and Mark A. Musen. Integration of temporal reasoning and temporal-data maintenance into a reusable database mediator to answer abstract, time-oriented queries: The Tzolkin system. *Journal of Intelligent Information Systems*, 13(1–2):121–145, 1999.

[48] Martin J. O'Connor, William E. Grosso, Samson W. Tu, and Mark A. Musen. RASTA: A distributed temporal abstraction system to facilitate knowledge-driven monitoring of clinical databases. In Vimla L. Patel et al., editors, *Proceedings of the 10th World Congress on Medical Informatics (Medinfo 2001)*, pages 508–512, London, UK, 2001. IOS Press.

[49] Martin J. O'Connor, Samson W. Tu, and Mark A. Musen. The Chronus II temporal database mediator. In Isaac Kohane, editor, *Bio*medical Informatics: One Discipline. Proceedings of the AMIA 2002 Annual Symposium*, pages 567–571, San Antonio, TX, 2002. Hanley and Belfus.

[50] Omolola Ogunyemi, Qing Zeng, and Aziz A. Boxwala. BNF and built-in classes for object-oriented guideline expression language (GELLO). Technical Report DSG-TR-2001-018, Brigham and Women's Hospital, Boston, MA, 2001.

[51] Gultekin Ozsoyoglu and Richard T. Snoodgrass. Temporal and real-time databases: a survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, August 1995.

[52] Mor Peleg, Aziz A. Boxwala, Omolola Ogunyemi, Qing Zeng, Samson W. Tu, Ronilda Lacson, Elmer Bernstam, Nachman Ash, Peter Mork, Lucila Ohno-Machado, Edward H. Shortliffe, and Robert A. Greenes. GLIF3: The evolution of a guideline representation format. In J. Marc Overhage, editor, *Converging Information, Technology, and Health Care. Proceedings of the AMIA 2000 Annual Symposium*, pages 645–649, Los Angeles, CA, 2000. Hanley and Belfus.

[53] Mor Peleg, Omolola Ogunyemi, Samson W. Tu, Aziz A. Boxwala, Qing Zeng, Robert A. Greenes, and Edward H. Shortliffe. Using features of Arden Syntax with object-oriented medical data models for guideline modeling. In Susan Bakken, editor, *Visions of the Future and Lessons from the Past. Proceedings of the AMIA 2001 Annual Symposium*, pages 523–527, Washington, DC, 2001. Hanley and Belfus.

[54] Mor Peleg, Samson W. Tu, Jonathan Bury, Paolo Ciccarese, John Fox, Robert A. Greenes, Richard Hall, Peter D. Johnson, Neill Jones, Anand

Kumar, Silvia Miksch, Silvana Quaglini, Andreas Seyfang, Edward H. Shortliffe, and Mario Stefanelli. Comparing computer-interpretable guideline models: A case-study approach. *Journal of the American Medical Informatics Association*, 10:52–68, 2003.

[55] Ian N. Purves, Bob Sugden, Nick Booth, and Mike Sowerby. The PRODIGY project – the iterative development of the release one model. In Nancy M. Lorenzi, editor, *Transforming Health Care Through Informatics: Cornerstones for a New Information Management Paradigm. Proceedings of the AMIA 1999 Annual Symposium*, pages 359–363, Washington, DC, 1999. Hanley and Belfus.

[56] Silvana Quaglini, Mario Stefanelli, Anna Cavallini, Giuseppe Micieli, Clara Fassino, and C. Mossa. Guideline-based careflow systems. *Artificial Intelligence in Medicine*, 20(1):5–22, September 2000.

[57] Silvana Quaglini, Mario Stefanelli, Giordano Lanzola, Vincenzo Caporusso, and Silvia Panzaras. Flexible guideline-based patient careflow systems. *Artificial Intelligence in Medicine*, 22(1):65–80, April 2001.

[58] Jean-Francois Rit. Propagating temporal constraints for scheduling. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 383–388, Menlo Park, California, 1986. AAAI Press.

[59] Andreas Seyfang. *An Integrated System for Temporal Data Abstraction to Facilitate Guideline Execution and Knowledge-Based Data Analysis.* PhD dissertation, Vienna University of Technology, Institute of Software Technology and Interactive Systems, Vienna, 2006. To appear, by courtesy of the author.

[60] Andreas Seyfang, Robert Kosara, and Silvia Miksch. Asbru reference manual, Asbru version 7.3. Technical Report Asgaard-TR-2002-1, Vienna University of Technology, Institute of Software Technology and Interactive Systems, Vienna, January 2002.

[61] Andreas Seyfang and Silvia Miksch. Advanced temporal data abstraction for guideline execution. In Katharina Kaiser, Silvia Miksch, and Samson W. Tu, editors, *Computer-based Support for Clinical Guidelines and Protocols. Proceedings of the Symposium on Computerized Guidelines and Protocols (CGP 2004)*, volume 101 of *Studies in Health Technology and Informatics*, pages 88–103, Prague, 2004. IOS Press.

[62] Andreas Seyfang, Silvia Miksch, Werner Horn, Michael S. Urschitz, Christian Popow, and Christian F. Poets. Using time-oriented data abstraction methods to optimize oxygen supply for neonates. In Silvana Quaglini, Pedro Barahona, and Steen Andreassen, editors, *Artificial Intelligence in Medicine. Proceedings of the 8th Conference on Artificial*

*Intelligence in Medicine in Europe (AIME 2001)*, pages 217–226, London, UK, 2001. Springer-Verlag.

[63] Andreas Seyfang, Silvia Miksch, and Mar Marcos. Combining diagnosis and treatment using asbru. *International Journal of Medical Informatics*, 68(1–3):49–57, 2002.

[64] Andreas Seyfang, Silvia Miksch, Cristina Polo-Conde, Jolanda Wittenberg, Mar Marcos, and Kitty Rosenbrand. MHB – a many-headed bridge between informal and formal guideline representations. In Silvia Miksch, Jim Hunter, and Elpida T. Keravnou, editors, *Artificial Intelligence in Medicine. Proceedings of the 10th Conference on Artificial Intelligence in Medicine in Europe (AIME 2005)*, pages 146–150, Aberdeen, UK, 2005. Springer-Verlag.

[65] Yuval Shahar, Silvia Miksch, and Peter Johnson. The asgaard project: a task-specific framework for the application and critiquing of time-oriented clinical guidelines. *Artificial Intelligence in Medicine*, 14(1–2):29–51, September 1998.

[66] Yuval Shahar and Mark A. Musen. Knowledge-based temporal abstraction in clinical domains. *Artificial Intelligence in Medicine*, 8(3):267–298, July 1996.

[67] Ravi D. Shankar and Mark A. Musen. Justification of automated decision-making: Medical explanation or medical argument? In Nancy M. Lorenzi, editor, *Transforming Health Care Through Informatics: Cornerstones for a New Information Management Paradigm. Proceedings of the AMIA 1999 Annual Symposium*, pages 395–399, Washington, DC, 1999. Hanley and Belfus.

[68] Yoav Shoham. Temporal logics in ai: Semantical and ontological consideration. *Artificial Intelligence*, 33(1):89–104, September 1987.

[69] Richard T. Snodgrass. The temporal query language tquel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[70] Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[71] Richard T. Snodgrass and Ilsoo Ahn. A taxonomy of time databases. In *SIGMOD '85: Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, pages 236–246, New York, NY, USA, 1985. ACM Press.

[72] Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen, and Andreas Steiner. Transitioning temporal support in TSQL2 to SQL3. In

Opher Etzion et al., editors, *Temporal Databases: Research and Practice*, pages 150–194. Springer-Verlag, 1997.

[73] Alex Spokoiny and Yuval Shahar. A knowledge-based time-oriented active database approach for intelligent abstraction, querying and continuous monitoring of clinical data. In Marius Fieschi et al., editors, *Proceedings of the 11th World Congress on Medical Informatics (Medinfo 2004)*, pages 84–88, San Antonio, TX, 2004. IOS Press.

[74] David R. Sutton and John Fox. The syntax and semantics of the proforma guideline modeling language. *Journal of the American Medical Informatics Association*, 10(5):433–443, September/October 2003.

[75] Paolo Terenziani, Carlo Carlini, and Stefania Montani. Towards a comprehensive treatment of temporal constraints in clinical guidelines. In *9th International Symposium on Temporal Representation and Reasoning (TIME '02)*, pages 20–27, Manchester, UK, 2002. IEEE Computer Society Press.

[76] Paolo Terenziani, Gianpaolo Molino, and Mauro Torchio. A modular approach for representing and executing clinical guidelines. *Artificial Intelligence in Medicine*, 23(3):249–276, 2001.

[77] Paolo Terenziani, Stefania Montani, Alessio Bottrighi, Mauro Torchio, and Gianpaolo Molino. Supporting physicians in taking decisions in clinical guidelines: the GLARE "what if" facility. In Isaac Kohane, editor, *Bio*medical Informatics: One Discipline. Proceedings of the AMIA 2002 Annual Symposium*, pages 772–776, San Antonio, TX, 2002. Hanley and Belfus.

[78] Paolo Terenziani, Stefania Montani, Alessio Bottrighi, Mauro Torchio, Gianpaolo Molino, and Gianluca Correndo. The GLARE approach to clinical guidelines: Main features. In Katharina Kaiser, Silvia Miksch, and Samson W. Tu, editors, *Computer-based Support for Clinical Guidelines and Protocols. Proceedings of the Symposium on Computerized Guidelines and Protocols (CGP 2004)*, volume 101 of *Studies in Health Technology and Informatics*, pages 162–166, Prague, 2004. IOS Press.

[79] Paolo Terenziani and Richard T. Snodgrass. Reconciling point-based and interval-based semantics in temporal relational databases: A treatment of the telic/atelic distinction. *IEEE Transactions on Knowledge and Data Engineering*, 16(5):540–551, 2004.

[80] Samson W. Tu and Mark A. Musen. From guideline modeling to guideline execution: Defining guideline-based decision-support services. In J. Marc Overhage, editor, *Converging Information, Technology, and*

*Health Care. Proceedings of the AMIA 2000 Annual Symposium*, pages 863–867, Los Angeles, CA, 2000. Hanley and Belfus.

[81] Samson W. Tu and Mark A. Musen. Modeling data and knowledge in the eon guideline architecture. In Vimla L. Patel et al., editors, *Proceedings of the 10th World Congress on Medical Informatics (Medinfo 2001)*, pages 280–284, London, UK, 2001. IOS Press.

[82] Michael S. Urschitz, Werner Horn, Andreas Seyfang, Antonella Hallenberger, Tina Herberts, Silvia Miksch, Christian Popow, Ingo Müller-Hansen, and Christian F. Poets. Automatic control of the inspired oxygen fraction in preterm infants – a randomized crossover trial. *American Journal of Respiratory and Critical Care Medicine*, 170(10):1095–1100, September 2004.

[83] Lluís Vila and Eddie Schwalb. A theory of time and temporal incidence based on instants and periods. In *3rd International Workshop on Temporal Representation and Reasoning (TIME '96)*, pages 21–28, Key West, FL, May 1996. IEEE Computer Society Press.

[84] Margret C. M. Vissers, Arie Hasman, and Cees J. van der Linden. Impact of a protocol processing system (ProtoVIEW) on clinical behaviour of residents and treatment. *International Journal of Biomedical Computing*, 42(1–2):143–150, July 1996.

[85] Peter Votruba, Silvia Miksch, and Robert Kosara. Facilitating knowledge maintenance of clinical guidelines and protocols. In Marius Fieschi et al., editors, *Proceedings of the 11th World Congress on Medical Informatics (Medinfo 2004)*, pages 57–61, San Antonio, TX, 2004. IOS Press.

[86] Dongwen Wang, Mor Peleg, Samson W. Tu, Aziz A. Boxwala, Omolola Ogunyemi, Qing Zeng, Robert A. Greenes, Vimla L. Patel, and Edward H. Shortliffe. Design and implementation of the GLIF3 guideline execution engine. *Journal of Biomedical Informatics*, 37(5):305–318, October 2004.

[87] Ohad Young and Yuval Shahar. The Spock system: Developing a runtime application engine for Hybrid-Asbru guidelines. In Silvia Miksch, Jim Hunter, and Elpida T. Keravnou, editors, *Artificial Intelligence in Medicine. Proceedings of the 10th Conference on Artificial Intelligence in Medicine in Europe (AIME 2005)*, pages 166–170, Aberdeen, UK, 2005. Springer-Verlag.