TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

Ph.D. Thesis

An Integrated System for Temporal Data Abstraction to Facilitate Guideline Execution

Conducted for the purpose of receiving the academic title
"Doktor der technischen Wissenschaften"

Supervisor
Silvia Miksch
Institute of Software Technology and Interactive Systems

Submitted at the Vienna University of Technology
Faculty of Informatics

by

Andreas Seyfang
8552129
Dr. Josef Reschplatz 1/21
A-1170 Vienna

Vienna, July 16, 2011

Abstract

Clinical guidelines and protocols are important means to improve the quality of care. To make their application more efficient, they are translated to computer-interpretable models (CIMs) using languages such as Asbru. One of the fundamental conditions for the success of such an application is the integration into the information flow at the place of care. This demands for temporal data abstraction, which bridges the low-level data from monitoring devices and laboratory results to the high-level concepts used in guideline or protocol.

Existing approaches mostly focus on low-frequency domains such as primary care. Consequently, I focus on high-frequency domains such as intensive care units which adds the requirement that the algorithms must be noise-resistant and time-efficient. Asbru was chosen as a starting point because its expressive power with respect to temporal data abstraction excels the alternative approaches.

From this results the main research question answered in this thesis: How can temporal data abstraction be combined with the execution of clinical guidelines and protocols in a fashion suitable for high-frequency domains? "Suitable for high-frequency domains" implies that continuously arriving data is handled in a timely manner irrespective of the potentially large history of measurement, and that noise is handled on many levels, because their combination promises the best success.

In this thesis, I present the following.

- A versatile abstraction algorithm called Spread to handle noise of varying intensity and to generate steady, qualitative values based on such input. It is based on the standard error or alternatively, the quantiles, for measurements in a moving time window of configurable size.
- An efficient implementation for matching parameter propositions with input data. Parameter propositions are at the core of patient monitoring in the Asbru language. They describe temporal constraints to be met by the measurements in order for some condition to be fulfilled.

  My algorithms replace the repeated queries of potentially large recordings of data used by existing solutions with multi-dimensional state machines representing the progress of the matching process for a given parameter proposition.
- Integration of all the involved modules for temporal data abstraction, monitoring, and plan execution into a seamless framework. This reduces system complexity while increasing the potential of the overall system due to the unlimited combination of modules.

- Mapping of Asbru to these modules, and mapping of the abstraction modules to Asbru. I show how to transform a CIM using Asbru into an assembly of abstraction modules for which the implementation has been described. In addition, I expanded the syntax of Asbru to describe novel abstractions, like the Spread.

These above was evaluated on a practical and on a theoretical level. On the practical level, ideas from this thesis are fundamental in two lines of application. On the one hand, the abstraction of steady qualitative values from noisy data permits the control of oxygen supply in a neonatal intensive care unit at the level of a human expert dedicated to the job, and superior to clinical routing, as shown in a clinical study. On the other hand, the integrated framework for temporal data abstraction, monitoring, and plan execution forms the basis of the Asbru interpreter which was (and is) used successfully in international research projects.

On the theoretical level, this thesis discusses the computational effort associated with the proposed algorithms, how they meet the objectives, and their benefits and limitations.

Although the field of application of the work described in this thesis is medicine, with a focus on intensive care, the described methods apply to data abstraction and plan execution in any field in which heterogeneous, time-oriented data must be matched with complex domain knowledge.

Zusammenfassung

Therapierichtlinien sind wichtige Mittel zur Qualitätssicherung in der Medizin. Um ihren Einsatz effizient zu gestalten werden sie in Modelle übersetzt, die durch Computer ausgeführt werden können. Dabei werden Modellierungssprachen wie z.B. Asbru verwendet. Eine wesentliche Voraussetzung für den Erfolg ist die Integration des Systems in den bestehenden Informationsfluß am Anwendungsort. Das wiederum erfordert zeitbezogene Datenabstraktion, die die Brücke schlägt zwischen den Rohdaten, die Meßgeräte und Laboruntersuchungen liefern, und den abstrakten Begriffen, die in Therapierichtlinien verwendet werden.

Existierende Ansätze beschränken sich auf Anwendungen wie sie in der Ordination eines praktischen Arztes anfallen, d.h. Anwendungen, die durch geringes Datenvolumen gekennzeichnet sind. Dementsprechend konzentriere ich mich auf sog. Hochfrequenzbereiche, in denen große Mengen von Daten in kurzer Zeit anfallen, wie z.B. Intensivstationen. Lösungsansätze in diesen Bereichen müssen robust in Bezug auf fehlerhafte Meßwerte sein, und sie müssen eingehende Daten rasch abarbeiten. Als Modellierungssprache für Therapierichtlinien wurde Asbru gewählt, weil es andere Sprachen an Ausdrucksmöglichkeiten bezüglich der zeitbezogenen Datenabstraktion übertrifft.

Daraus folgt die Forschungsfrage, die in dieser Dissertation beantwortet wird: Wie kann zeitbezogene Datenabstraktion mit der Ausführung von Therapierichtlinien so verknüpft werden, daß es Hochfrequenzbereichen entspricht? Das heißt, daß kontinuierlich gelesene Daten in kurzer Zeit verarbeitet werden müssen, auch wenn die Ausführungseinheit lange in Betrieb ist und daher die historischen Daten einen Patienten betreffend große Ausmaße annehmen. Außerdem müssen Signalstörungen auf verschiedenen Ebenen aufgefangen werden, weil nur die Kombination verschiedener Ansätze das beste Ergebnis verspricht.

In dieser Dissertation werden die folgenden Lösungsansätze beschrieben:

- Ein vielseitiger Abstraktionsalgorithmus namens Spread, der aus numerischen Eingaben mit wechselndem Anteil an störenden Schwankungen kontinuierliche qualitative Werte generiert. Er beruht auf dem Standard Error (oder alternativ dazu den Quantilen) von Meßwerten innerhalb eines frei definierbaren Zeitfensters, das mit fortlaufender Messung weitergeschoben wird.

- Eine effiziente Implementierung des Abgleichs von Parameter Propositions mit Eingabedaten. Parameter Propositions sind das Herzstück der Überwachung des Patientenzustandes in Asbru. Sie beschreiben zeitbezogene Bedingungen, die Meßwerte erfüllen müssen, um eine bestimmte, in der Therapierichtlinie definerte, Bedingung zu erfüllen.

Mein Algorithmus ersetzt die wiederholten Abfragen von potentiell großen Datenbeständen durch einen mehrdimensionalen endlichen Automaten, der den Fortschritt beim Abgleich einer bestimmten Parameter Propositions mit schrittweise verfügbar werdenden Eingabedaten modelliert.

- Integration aller Module für die zeitbezogen Datenabstraktion, die Überwachung des Patientenzustandes und die Ausführung der Therapierichtlinie in einem einheitlichen Gesamtsystem. Dadurch wird die Systemkomplexität reduziert und gleichzeitig durch die freie Kombinierbarkeit aller Module die Problemlösungskapazität erhöht.

- Die Abbildung des Funktionsumfangs von Asbru auf die zuvor genannten Module. Ich zeige wie Therapierichtlinien, die in Asbru formuliert sind, in ein Netzwerk von Abstraktionsmodulen übergeführt werden, und beschreibe die Implementierung dieser Module. Zusätzlich erweiterte ich den Sprachumfang von Asbru, um neue Abstraktionsalgorithmen wie den Spread in Asbru parametrisieren zu können.

Die in dieser Dissertation beschriebenen Ansätze wurden auf praktischer und theoretischer Ebene evaluiert. Auf praktischer Ebene bilden sie die Grundlage für zwei Anwendungen. Einerseits erlaubt die Gewinnung kontinuierlicher qualitativer Werte aus teilweise fehlerbehafteten Meßwerten die Steuerung der Sauerstoffzufuhr in einer Frühgeborenenintensivstation, gleich gut wie durch einen Experten der sich ausschließlich dieser Aufgabe widmet, und besser als im Alltag, wie durch eine klinische Studie gezeigt wurde. Andererseits bilden die hier beschriebenen Module zur zeitbezogen Datenabstraktion, Überwachung des Patientenzustandes und Ausführung der Therapierichtlinie die Grundlage des Asbru Interpreters, der in mehreren internationalen Forschungsprojekten erfolgreich eingesetzt wurde bzw. wird.

Auf theoretischer Ebene beschreibt diese Dissertation die Algorithmenkomplexität sowie die Vorteile und Grenzen der einzelnen Teilösungen, und wie sie den gesetzten Zielen entsprechen.

Obwohl das Anwendungsgebiet der hier beschriebenen Ideen die Medizin, insbesondere die Intensivmedizin ist, können die beschriebenen Lösungen ohne weiteres auf andere Gebiete übertragen werden, in denen heterogene, zeitbezogene Daten und komplexes Wissen über das Anwendungsgebiet zusammentreffen.

Contents

Chapter 1

Introduction and Motivation

1.1 Domain description

The field of application of the work described in this thesis is medicine, with a focus on intensive care. However, the described methods apply to data abstraction and plan execution in any field in which heterogeneous, time-oriented data must be matched with complex knowledge.

In the field of medicine, the application of clinical guidelines and protocols helps to improve the quality of care by ensuring the optimal choice of treatment. Computer-supported guidelines reduce the workload of professionals by presenting the necessary information at the point of care. In addition, guidelines reduce expenses for drugs and other material by eliminating unnecessary examinations or treatment phases. This is described in detail in Section 1.1.1.

An indispensable precondition for the successful application of clinical guidelines and protocols is their integration into clinical practice. Besides other issues such as workflow organisation, hardware compatibility and data access procedures, and the need to create a computer-interpretable model of the guideline or protocol, temporal data abstraction is required to automatically map time-annotated raw-data (e.g., percent of oxygen in blood at a certain second) to high-level medical concepts (e.g., sufficient oxygen saturation during an extended period of observation). This is performed by temporal data abstraction. Section 1.1.2 gives more details on this field.

The procedural knowledge about the treatment, and the current state of the treatment, forms an important context for temporal data abstraction. The same numeric value delivered by a monitoring device, e.g., an oxygen saturation of the blood measured by a pulsoximeter device, can mean very different things, depending on the context of the treatment. Under artificial ventilation, a saturation value of 99% is considered very high, potentially leading to damage of the retina if maintained over extended periods of time. Under normal, natural ventilation, the same amount of oxygen saturation of the blood is normal and has no negative consequences for the health. This means that the abstraction of the value 99 to the medical concept of "too high" versus "normal" depends on the context which is constituted by performing artificial ventilation or not.

This means that temporal data abstraction and guideline execution form a symbiotic pair:

- Guideline execution needs temporal data abstraction to be embedded into the data flow at the point of care, thus providing the right recommendation at the right time without additional user input.

- Temporal data abstraction needs guideline execution to provide the context of abstraction. Without the background formed by the current treatment mode and general patient situation, it is impossible in many situations to generate meaningful medical concepts from the raw data.

The preferred field of application for this work are *high-frequency domains* such as intensive care units. In this context, high frequency refers to 1 Hz to 1 kHz; and low-frequency refers to measurements lieing many hours, if not days, appart. Typical examples are intensive care for high-frequency domain; and diabetes monitoring for for low-frequency.

To handle this type of high-frequency data, implementations must have near real-time characteristics – they must react quickly enough to the input to ensure processing at the quoted rate, but strong guarantees that 100 % of all computation paths complete within a constant time frame are not necessary.

1.1.1 Guidelines and decision support

There is a range of different but related terms and entities associated with guidelines and decision support.

Clinical guidelines – also called practice guidelines – are "systematically developed statements to assist practitioner and patient decisions about appropriate health care for specific clinical circumstances" [54, p. 38]. They focus on important questions in a specific field, not necessarily providing complete guidance to a certain medical task, but complementing the available text books. Today, most clinical guidelines are based on clinical evidence [64].

Protocols are instructions for the staff at a particular site of care, e.g., a particular hospital [114, 34]. Many of them are based on clinical guidelines. They provide more concise descriptions of actions and conditions, and generally cover a certain medical task from start to end.

Computer-interpretable models of guidelines and protocols (CIMs) are representations of clinical guidelines or protocols in a formal representation language, such as Asbru, GLIF, Guide or ProForma [106].

Clinical decision support systems (CDSS) Clinical Decision Support Systems are "active knowledge systems which use two or more items of patient data to generate case-specific advice" [177]. Systems executing CIMs are seen as CDSS, e.g., in [48].

Reminder systems can be implemented based on CIMs driven by timers, but also by data entered into an EPR. They send messages to care staff or patients. In the first case, they differ from DSS only in their rather reactive nature. In the second case, they improve compliance in fields like diabetes monitoring, diet planning or smoking cessation.

Trial alert systems can be seen as a subgroup of reminder systems. They compare entered patient data against eligibility criteria of clinical trials and suggest recruitment as appropriate [51].

Besides improving the quality of care, clinical guidelines and protocols also reduce costs. It has be proved that adherence to guidelines and protocols may reduce healthcare costs by up to 25% [32]. Computer-support is required to improve daily care practice, distributing paper versions of the guideline alone is not sufficient [45].

There are many flavours of CIMs, not only regarding their deployment, but also regarding the degree to which the original text is transformed. Simple approaches restrict themselves to classifying guidelines and symptoms on the level of keys [154]. The main advantage is that the effort for translation of the original guideline into such an annotated free text form is relatively small. The disadvantage of this approach is that the computer only performs limited search support, but resorts to the user for any decision, even simple ones, since the data is not available, nor modelled in the system.

The more advanced alternative to such a solution is the modelling of a guideline or protocol in a completely computer executable form, such as Asbru [125]. While this causes considerable modelling effort, it allows the system to select appropriate treatment steps without user intervention (except where desired). The work described in this thesis builds on CIMs modelled in Asbru.

Section 2.1 describes the related work in this field including details on Asgaard and Asbru. It also justifies the decision to use Asbru as a basis for this thesis.

1.1.2 Temporal data abstraction

Data abstraction is the process of mapping low-level raw-data to high-level concepts. In the medical domain, typical examples for raw data are the mostly numerical (i.e., quantitative) results delivered by various laboratory tests or monitoring devices in an intensive care unit. Examples for high-level concepts, or pieces of information, are *high fever* and *sufficient oxygen*.

The mapping between the data and information is context dependent – the same percentage of oxygen in the blood can be normal in one context and health-threatening in another context. The context is formed by features such as age of patient, underlying diseases, and current mode of treatment.

The field of *temporal* data abstraction deals with the abstraction of measurement series taken over a certain time span. Analyzing the temporal dimension is important for both long term evaluation of a treatment and short term detection of changes in the patient's state. Furthermore, many high-level concepts contain temporal aspects in their specification, such as "serious fever is high fever lasting for more than 12 hours". We therefore cannot abstract information about the presence of serious fever without considering the temporal aspect of the measurements.

Section 2.2 describes the related work in the field of temporal data abstraction.

An important hurdle to overcome in this abstraction process is noise. Merriam-Webster defines noise (among other things) as "an unwanted signal or a disturbance (as static or a variation of voltage) in an electronic device or instrument (as radio or television)" or "irrelevant or meaningless data or output occurring along with desired information" [4].

In the context of computer applications in medicine, it is an unexplained distortion of the measurement. The reason is often a combination of technical difficulty and physiological limitation. E.g., measuring the oxygen saturation in the blood using a sensor in the artery would deliver perfect data, but the health implications of setting such a sensor cannot be justified in many cases. A pulsoximeter device placed on the finger or toe measures the light refraction of oxygen-saturated and unsaturated haemoglobin through the skin without hurting it or causing discomfort, but its measurements are unstable.

This is but one example where considerations of care practice result in a suboptimal setting for data processing. Put in a positive way, advances in data processing remove the negative side effects of older measuring technology without reducing the level of care.

1.1.3 Integration

An important aspect of CIM execution is the integration into the data flow at the point of care [74].

A cluster randomized controlled trial in the field of screening and treatment of dyslipidemia carried out in 38 Dutch general practices compared automatically generated alerts with on-demand decision support [178]. In those practices receiving alerts based on the data entered into the electronic health record as part of daily practice, 65% of the patients requiring screening were screened and 66% of patients requiring treatment were treated. For those practices where the user had to active access the recommendations, 35% of the patients requiring screening were screened and 40% of patients requiring treatment were treated. In the control group without electronic decision support, 25 % of the patients requiring screening were screened and 36% of patients requiring treatment were treated.

A systematic review of trials to identify features critical to success, analysing 70 studies, four the following four features to be independent predictors of improved clinical practice: automatic provision of decision support as part of clinician workflow, provision of recommendations rather than just assessments, provision of decision support at the time and location of decision making, and computer deployment in decision support [85].

In a review of 100 controlled trials on practitioner performance or patient outcomes, improved practitioner performance was associated with (1) automatically prompting users compared with requiring users to activate the system, and (2) studies in which the authors also developed the DSS software compared with studies in which the authors were not the developers [57]. While the latter can be seen as either bias or improved integration, the former clearly supports the findings above.

From the technical perspective, integration must be achieved on three different levels:

1. Hardware must lay first foundations for the data flow. An example, related to the work presented here, is the communication between the laptop running our system, the pulsoximetry device measuring the oxygen saturation in the blood of the patient, and the respirator, controlling the oxygen supply for the patient. It took years to find a ventilator manufacturer willing to provide a serial connection (or any connection) to an external computer. Today, our solution is considered

for integration into a respirator device. Further details on this project are given in Section 5.1.1.

2. Software interfaces must bridge the DSS to the electronic patient record and other relevant information. The link from this thesis to work on this level is provided by my participation in the projects OncoCure[1] [47] and Remine[2]. In the former, the Asbru interpreter [125] (implementing the ideas described here) is integrated with the electronic patient record system at an Italian hospital. In the latter, the Asbru interpreter is integrated with a comprehensive database system which aims at capturing all information needed to reduce adverse events at four different hospitals in three countries. Further details on both projects are given in Section 5.1.2.

   There is comprehensive work on binding the data-item references in a CIM to controlled vocabularies and ontologies, which is beyond the scope of this thesis.

3. Temporal data abstraction algorithms must transform the low-level data to high-level concepts. This is the main focus of this thesis.

Besides all the technical solutions to various challenges, there are many important factors for success which are also clearly documented in the literature.

- The persons dealing with the system must see some personal benefit in using it; Or they need to feel pressure from close collaborators who see direct benefits from the system [111]. Involving users in early phases of the design process increases their willingness to use the system later in their daily routine. Users only use a system if they believe that they need it, before trying it [170].

- There must be support from the organization. If the decision makers in an organization are decided to make a new system a success, because they see a strong benefit for the organization, then initial obstacles are overcome and the benefit of the system can be felt before disappointment can set in [111, 115, 182]. Organizational changes associated with the introduction of DSS can be significant and need attention [82].

- The time available to physicians to reason about recommendations seems to be a factor, too. A survey of factors affecting clinician acceptance of clinical DSSs showed that respondents had a generally positive attitude towards DSSs, but 80% state that they were less likely to accept alerts when they were behind schedule and 84% admitted to being at least 20 minutes behind schedule at least some time [144]. Embedded explanation modules linking the recommendation to all relevant pieces of the underlying guidelines with minimal user intervention could help to overcome such barriers.

[1] `http://www.donau-uni.ac.at/en/department/ike/forschung/planmanagement/projekte/11128/index.php`, last accessed May 5th, 2011

[2] `www.remine-project.eu`, last accessed Oct 20th, 2010

1.1.4 Importance of the field

In a recent commentary on the past, present and future [69], two views on important future research fields are given. In the traditional view, "computer-enhanced decision-support for health care professionals, combined with appropriate concepts for reasoning and knowledge representation" [69, p. 606] is one of ten important research fields, and electronic patient records another. In the revolutionary view, five of 16 items relate to work presented here. They are:

1. "seamless interactivity with automated data capture and storage for patient care, and beyond (from perception to high-level semantic concepts, related to human-human, machinemachine, as well as humanmachine interaction; beyond in the meaning of not being restricted to certain disease episodes);" [69, p. 606]

2. "knowledge-based decision-support for diagnosis and therapy, and beyond (with decision-support in its broadest meaning, i.e. from simply pointing persons to important knowledge by identifying latest results in knowledge bases to context-aware, individualized decision proposals using formally represented knowledge; beyond in the meaning of also including, e.g. prevention);" [69, p. 606]

3. "patient-centered data analysis and mining (with representations of patient data based on appropriate semantic concepts);" [69, p. 606]

4. "informatics diagnostics, where informatics tools (with corresponding methodology) form the major part of the diagnostic entity;" [69, p. 606]

5. "informatics therapeutics, where informatics tools (with corresponding methodology) form the major part of the therapeutic entity;" [69, p. 606]

Item 1-3 directly relate to the close integration of data abstraction and treatment planning. Item 4 refers to image processing as well as to sophisticated temporal data abstraction. For clinical decision support systems, item 5 may not match. For advisory systems it might. Another item of the 16 is "automated, individualized health advice and education;" [69, p. 606] which is an alternative application to the same framework (presented here). With a text generation module or a nice graphical interface, the recommendations generated by the DSS can be mapped to recommendations addressed at the patient. The fully automated nature of the framework presented here makes is most suitable for scalable advisory systems where many users receive advise without expert intervention. Fields like diabetes involve both complex and time-oriented data abstraction; and the demand for automatic patient reminders or encouragement, combined with alarming of human experts in case of serious non-compliance.

At the same time, we must be aware that even simple changes to user interfaces can seriously impact user performance, however sophisticated or simple the system behind it might be. In a prospective study, the subset of tests in a computerized provider order entry system which can be ordered via a simple checkbox was altered. The newly added tests saw an increase of 60% and 90% in the following two years. The orders which needed more than ticking a checkbox after the change saw a decrease of 27% and 19%. This group constituted 50% of the tests on the initial list. The development of tests remaining on the list of checkboxes was stable. However, the decision which tests to remove and which to add was based on clinical best practice recommendations. This means that part of the change could be attributed to physicians increasingly following these guidelines, independently of the data input format [138].

1.2 Research Question

In the following, I present the research question and its subquestions.

1.2.1 Main Question

The main research question answered in this thesis is:

> How can temporal data abstraction be combined with the execution of clinical guidelines and protocols in a fashion suitable for high-frequency domains?

Suitability is defined by efficient processing of input data as explained below.

1.2.2 Sub Questions

The main research question can be broken into the following subquestions.

> **Subquestion 1:** How can short response times be established for arbitrarily high volumes of data?

The domain of application (intensive care medicine) does not demand for true real-time behaviour like the control of engines does. However, it is crucial that the response time does not depend on the total length of historic recordings, and that it is a fraction of a second in the general case.

> **Subquestion 2:** How can steady values be abstracted from qualitative input with varying amounts of noise and gaps of varying length?

Noise and missing data is one of the dominant problems in medicine. While statistics provides some solutions to deal with both, the medical domain demands knowledge-based solutions which take into account the role of a signal in the decision process. A solution which may be perfect for one input can be unacceptable for another. Furthermore, sources of disturbances, like patient movement or care interventions result in large variations of noise and/or fraction of missing data. Again, the domain knowledge demands special solutions ranging from waiting for the end of the disturbance to graceful degradation.

> **Subquestion 3:** How can the execution of Asbru plans be combined with temporal data abstraction in an efficient way?

In this thesis, Asbru is used as an example of a guideline modelling language. The choice is explained in Section 2.1.3. The question is, if and how the semantics of this language can be combined with time-efficient data abstraction, meeting the constraints sketched above.

1.3 Approach

This section describes the reasoning behind the individual chapters of my thesis, and the approach I took to solve the given problems and to answer the research question.

The scientist builds in order to study;
the engineer studies in order to build.

[24, p. 62]

Brooks, who stated the above, argues that computer science is an engineering discipline. Others see it as part of social sciences. In this thesis, I try to cover both views as far as possible, by describing what I built and what I studied in what I built.

The obvious answer of an engineer to a question like "How can temporal data abstraction be combined with the execution of clinical guidelines and protocols?" is to design such a system.

The scientist then explores whether it is a good and useful solution. This is done along two dimensions – rigor and relevance. Fällman and Grönlund define rigor as "a structured and controlled way of planning, carrying out, analyzing, evaluating and producing products of research, independently of the research method used", and relevance as "the act of making efforts into research issues that is of concern to a perceived audience" [53]. They point out that both depend on the intended audience.

For this thesis, the intended audience are computer scientists interested in further development of applied medical informatics. For the individual ideas contained in it, the audience are physicians as far as application in clinical practice is concerned, and computer scientists as far as publication is concerned.

The rigor of my work is established by a clear path from problem statement and the discussion of related work, to defining objectives, presenting a solution, and evaluating it. The relevance of my work is documented by its practical application and being accepted in scientific media.

To learn about the state of art, I read journals like International Journal of Medical Informatics, Artificial Intelligence in Medicine, and Journal of the American Medical Informatics Association and browsed the websites of colleagues working in the field to find additional publications. In addition, I attended conferences like European Conference on Artificial Intelligence (ECAI), Conference on Medical Informatics (MedInfo), and European Conference on Artificial Intelligence in Medicine (AIME); and workshops on Intelligent Data Analysis in Medicine and Pharmacology (IDAMAP) and on computerised guidelines (under various names). Section 2 describes those parts of research which directly relate to my thesis.

Based on this, and on the problem description detailed in Section 3, I developed objectives, and solutions to meet them. The guiding principles common to all partial solutions were that the solution must meet a clearly defined demand and that it must be novel enough to be of interest for the computer science community.

The work described in this thesis consists of the following lines of development.

Steady qualitative abstractions of noisy quantitative data. Data from monitoring devices is noisy. Existing approaches mostly use fixed thresholds or rules to eliminate errors and more extreme noise. However, for the purpose of artificial ventilation in neonates, we needed a method to map such input to steady qualitative values in such a fashion that smooth input leads to quick reaction to changes and noisy input leads to retaining the status-quo, and – most importantly – there is a smooth transition between the two extremes, i.e., medium noisy input leads to medium response time to changes.

The basic idea was to represent the measurements in a band (or *Spread*) of varying width, with the width depending on the amount of oscillation, because this resembled the physician's perception of the signal plot. The soundest measure for such deviations we found was the standard error, calculated for a linear regression of the measurements in a time window containing the last five minutes of measurements. The interval at which the input was re-evaluated, the size of the time window, the limits outside which input was discarded, and the minimum amount of remaining valid measurement per minute was defined by physicians in a series of meetings.

We evaluated the approach on previously recorded real patient data and found the signal asymmetric – decreased from the middle region were far more pronounced than increases. We therefore replaced the standard error plotted up and down from the centre by the 10% quantiles. The new version performed satisfactory in clinical trials, first in an open-loop setting, then in closed loop. Details are described in Section 5.1.1. Currently, the system is tested in a multi-centre study in three hospitals and a company is interested in integrating it into their respirators.

Statistical analysis of sliding time windows. The *Spread* is but one of a series of possible uses of a sliding time window which aggregates the measurement of a certain history of recent measurements. Figure 3.1 illustrates the necessity to slide this time window in small steps along the time axis instead of aggregating fixed intervals which introduces arbitrary separation into the input data which again leads to distortions.

Aggregating high-frequency input into intermediate formats with a lower frequency reduces computational load and eases the perception by the human.

Discussions with experts in intensive care showed that only few signals need to be monitored with minimal time delay, and that these monitoring tasks are carried out by commercial devices and additional alarms by our system would not be appreciated.

We therefore focused on high-quality abstractions to gain deeper insight into changes in the patient state.

Unfortunately, the practical hurdles of performing clinical studies did not permit us to evaluate more advanced ideas in clinical practice. As a consequence, I implemented these abstractions as a toolbox ready to use when a practical application requires them. The modular architecture of the system makes it easy to add modules if necessary.

Online-algorithms for the detection of temporal patterns. Asbru contains powerful means to describe the temporal constraints in input data. The measured data must be matched with these complex constraints. When a matching episode is found, the corresponding condition of an Asbru plan is fulfilled and the plan state changes.

Existing implementations were based on repeated queries of databases, and thus not suitable for high-frequency domains. A suitable solution had to be based on a finite-state machine. Analysing the complex semantics of the Asbru time annotation, I designed such multi-dimensional state machines – different variants of Asbru elements necessitated different state machines.

The algorithm was tested against the test data generated from the semantics definition of the time annotation. Section 5.2.5 shows the high efficiency of the solution. The relevance is given by the relevance of Asbru. Section 2.1 shows why Asbru is the best choice for combining temporal data abstraction with guideline execution in high-frequency domains.

Utility functions and an integrated framework. Studying the literature, I found many accounts on special abstraction algorithms for a given purpose. However, discussions of real-world problems showed that all such algorithms must become modules in a framework where the output of one module can be the input of another, and that simple things like arithmetic or logical combinations are absolutely mandatory components of such a toolbox.

In many cases, the best way to abstract raw data to medical concepts is not known in advance, but must be tried out in the knowledge acquisition process. To this end, many different abstractions must be performed side by side. It is therefore desirable that the code implementing the abstraction algorithm of a module is not called for every time step, but only if new input is available for this module. This is a prerequisite for benefiting from the reduction in data volume achieved by time-window-based abstraction and temporal pattern matching.

Therefore, I designed a central management unit which only enacts the necessary abstraction modules and keeps the others dormant. I also designed a hierarchy of time-annotated data-points, some of which represent measurements, some complex intermediate abstraction results like linear regression lines. Feature-extraction modules, while trivial by themselves, permit the conversion of complex abstractions back to simple values, e.g., to feed the slope of a regression line as a number into another abstraction module.

While these solutions are standard software engineering practice, they multiply the potential of the other parts of the solutions, and for any real-world application, they are indispensable. This is illustrated in [126].

Integrating plan execution. Asbru plans define the conditions under which the plan state changes in a declarative way, specifying a frequency at which the conditions are to be re-evaluated. In order to make the execution of Asbru plans ready for high-frequency domains, I ventured to remove the need for such resource-wasting active polling.

The original idea was to have a monitoring unit as a mediator between the data abstraction unit and the plan execution unit. It would receive requests to monitor from the plan execution unit. The latter would cancel these requests as conditions become irrelevant due to plan state changes effected by other circumstances. This called for extensive book keeping effort.

After the successful conversion of the declarative, retrospective definition of temporal constraint monitoring to state machines mentioned above, I tried with success to convert the Asbru plan semantics to state machines driven by the state of the plan conditions. Under this scheme, parent-child (i.e., plan-subplan) relations are implemented as a stream of complex (time-stamped) data-points. This makes the solution compatible with the unified abstraction framework, into which I already integrated the monitoring of temporal constraints before.

This integration expands the capabilities of the system even further, permitting the feedback from plan execution into data abstraction. Such cycles introduce difficulties, which do not exceed the difficulties introduced by the bi-directional communication between parent plan and child plan. Section 5.2.6 discusses the theoretical and practical impact of both sources of difficulties.

Mapping to Asbru. Since the above mentioned modules (state machines) to implement are designed to implement guideline models formalised in Asbru plans, a mapping of the syntax elements to the modules must be provided, but in theory (in Section 4.7) and in practice by making it part of the Asbru interpreter.

Following the principle to streamline model specification, I expanded the Asbru syntax [125] to cover data abstraction definitions in addition to plans.

This is a prerequisite for the efficient deployment of the above solutions – the elegance and usefulness of a seamless execution framework would be lost if not all of these modules could be described in a single configuration file, which is the Asbru plan library in this case.

These partial solutions form the Asbru interpreter, which is used in three international research projects and published in various scientific media as detailed in Section 1.4.

This thesis describes the following details on rigor and relevance in Section 5.

- The practical use of the *Spread* to control artificial ventilation in clinical studies.
- The practical deployment of the Asbru interpreter in scientific projects.
- The very favourable computational complexity of the proposed algorithms.
- How the proposed solutions meet the objectives set forth in Section 3.
- The description of the current limitations of these solutions, and how to overcome them.

This is rounded off by pointers at future work and a conclusion.

1.4 Dissemination

In the following, I describe the scientific articles in which the work described in this thesis is presented to various communities.

1.4.1 First ideas on the system architecture

The original idea was to have three entities, the data abstraction unit, the guideline execution unit, and the monitoring unit mediating between them. (Compare Section 4.1.) It is reflected in the following papers.

- S. Miksch and A. Seyfang. Continual planning with time-oriented, skeletal plans. In Proceedings of the 14th European Conference on Artificial Intelligence (ECAI 2000), pages 511-515, Berlin, 2000. IOS Press.

  In this paper, we take an AI perspective, using terminology from the planning community, and stressing the need to continually monitor patient state through treatment plan execution.

- A. Seyfang and S. Miksch. Integrating diagnosis and treatment in a flexible way. In R. Bellazzi and B. Zupan, editors, The Sixth Workshop on Intelligent Data Analysis in Medicine and Pharmacology (IDAMAP-2001) in Conjunction with the Conference on Medical Informatics (MedInfo 2001), 2001.

  In this workshop paper, we present our ideas based on an example from the field of artificial ventilation.

- A. Seyfang and S. Miksch. Modelling Diagnosis and Treatment, in Second Workshop on Computers in Anaesthesia and Intensive Care: Knowledge-Based Information Management, in Conjunction with the European Conference on Artificial Intelligence in Medicine (AIME 2001), Cascais, Portugal, pp. 20–24, 2001.

  This presents our ideas to the AI in Medicine community, using an example from a guideline for the management of chronic cough.

- S. Miksch, A. Seyfang, and R. Kosara. Plan management: Supporting all steps of protocol development and deployment. In EUNITE Workshop on Intelligent Systems in Patient Care, Vienna, 2001.

  Here, we describe our ideas on plan execution and data abstraction as part of a bigger picture comprising all steps from modelling to execution of clinical protocols.

- A. Seyfang, S. Miksch, and M. Marcos. Combining diagnosis and treatment using Asbru. Proceedings of the Conference on Medical Informatics (MedInfo 2001), pp. 533–537, 2001.

  In this paper, we describe the combination of data abstraction (for diagnosis modelling) and plan execution (for treatment modelling) from a medical informatics perspective.

- A. Seyfang, S. Miksch, and M. Marcos. Combining diagnosis and treatment using Asbru. International Journal of Medical Informatics, 68(1–3):49-57, 2002.

  This article is the revised version of the above conference presentation.

- A. Seyfang and S. Miksch. Advanced temporal data abstraction for guideline execution. In Symposium on Computerized Guidelines and Protocols (CGP 2004), pages 88-102. IOS Press, 2004.

  Here, we take a slightly different view at the system architecture, discussing examples from artificial ventilation and diabetes.

- S. Miksch, A. Seyfang, W. Horn, C. Popow, and F. Paky. Methods of temporal data validation and abstraction in high-frequency domains. In K. Cios, editor, Medical Data Mining and Knowledge Discovery, pages 320-357. Springer, 2001.

  Early ideas on temporal data abstraction and validation are described in this book chapter.

1.4.2 Implementation of the Asbru interpreter

The ideas presented in this thesis were implemented by Peter Votruba, Micheal Paesold and Gilbert Wondracek under my guidance. The resulting system is called the Asbru interpreter (compare Section 5.1.2 for the technical description and 5.1.2.1 for the description of the project funding this work). We presented the implemented system in the following publications.

- P. Votruba, A. Seyfang, M. Paesold, and S. Miksch. Environment-driven skeletal plan execution for the medical domain. In 17th European Conference on Artificial Intelligence (ECAI 2006), pages 847-848. IOS Press, 2006.

- P. Votruba, A. Seyfang, M. Paesold, and S. Miksch. Improving the execution of clinical guidelines and temporal data abstraction in high-frequency domains. In AI Techniques in Healthcare: Evidence-based Guidelines and Protocols, workshop in conjunction with ECAI 2006, pages 112-116, 2006.

- A. Seyfang, M. Paesold, P. Votruba, and S. Miksch. Improving the execution of clinical guidelines and temporal data abstraction in high-frequency domains. In A. ten Teije, S. Miksch, and P. Lucas, editors, Computer-based Medical Guidelines and Protocols: A Primer and Current Trends, pages 978-971, Amsterdam, 2008.

1.4.3 The Spread algorithm and its application

The *Spread* algorithm (compare Sections 4.4.4 and 5.1.1) and its application to control the artificial ventilation of preterm neonates was presented in the following set of publications.

- S. Miksch, A. Seyfang, W. Horn, and C. Popow. Abstracting steady qualitative descriptions over time from noisy, high-frequency data. In Artificial Intelligence in Medicine (AIME 1999), pages 281-290, Berlin, 1999. Springer.

This describes the first version of the *Spread* algorithm.

- A. Seyfang, S. Miksch, W. Horn, M. S. Urschitz, C. Popow, and C. F. Poets. Using time-oriented data abstraction methods to optimize oxygen supply for neonates. In Artificial Intelligence in Medicine (AIME 2001), pages 217-226, Berlin, 2001. Springer.

  The application of the further developed algorithm to online control of oxygen supply in neonates is described in this paper.

- M. S. Urschitz, V. Von Einem, A. Seyfang, and C. F. Poets. Use of pulse oximetry in automating O_2 delivery to ventilated infants. In STA-ISAPO Meeting Supplement to Anesthesia and Analgesia 94, 2002.

  This presents our approach from a medical perspective to neonatologists.

In addition, our project was introduced to the medical community by the following two abstracts.

- M. Urschitz, C. Poets, A. Seyfang, S. Miksch, W. Horn, and C. Popow. Konzept einer automatischen Anpassung des Sauerstoffbedarfs an die speziellen Bedürfnisse kleiner Frühgeborener. Monats-schrift für Kinderheilkunde, 149(4), 2001.

- M. Urschitz, C. Poets, A. Seyfang, S. Miksch, W. Horn, and C. Popow. Konzept einer automatischen Anpassung des Sauerstoffbedarfs an die speziellen Bedürfnisse kleiner Frühgeborener. Pneumologie, 55(4), 2001.

The results of the clinical study evaluating our application of the Spread algorithm were presented in the following publications.

- M. S. Urschitz, W. Horn, A. Seyfang, A. Hallenberger, T. Herberts, S. Miksch, C. Popow, I. Müller-Hansen, and C. F. Poets. Automatic control of the inspired oxygen fraction in preterm infants, a randomized cross-over trial. American Journal Respiratory and Critical Care Medicine (AJRCCM), 170, pages 1095-1100, 2004.

- S. Miksch, C. Popow, A. Seyfang, A. Hallenberger, M. Urschitz-Duprat, M. Urschitz, W. Horn, I. Müller-Hansen, C. Poets. "Klinische Evaluation einer automatischen FiO_2-Regelung bei beatmeten Frühgeborenen"; Poster: GNPI 2003, Köln; 2003; in: "29. Jahrestagung der Gesellschaft für Neonatologie u. Pädiatrische Intensivmedizin", (2003).

1.4.4 Bridging to the patient record

One of the prerequisites for the deployment of the work described in this thesis is the integration with the electronic patient record at the point of care. Our work on bridging the Asbru model to a real-world electronic patient record as part of the OncoCure project (compare Section 5.1.2.2) is presented in the following publications.

- C. Eccher, A. Seyfang, A. Ferro, and S. Miksch. Bridging an Asbru protocol to an existing electronic patient record. In Workshop on Knowledge Representation for Health-Care: Patient Data, Processes and Guidelines (KR4HC) in conjunction with AIME 2009, 2009.

- C. Eccher, A. Seyfang, A. Ferro, and S. Miksch. Embedding oncologic protocols into the provision of care: The Oncocure project. In The XXII International Conference of European Federation for Medical Informatics (MIE 2009). IOS Press, 2009.

1.5 Conventions

In this document, the following conventions are used.

- The word *I* is used to talk about my personal achievements while *we* is used when describing the group effort of the team of which I am a member.
- Throughout this thesis, the words *temporal* and *time-oriented* are used synonymously.
- To improve readability, I use the word *guideline* to replace abbreviations CIM (Computer-Interpretable Model) and CGP (Clinical Guideline or Protocol), whenever possible without loss of clarity.
- In Section 4, Asbru elements are set in *italics* while abstraction module names are set in sans serif font.
- This thesis does not contain references to the gender of an unknown person taking on a certain role. Such reference was however not removed in two cases where the gender of the individual concerned is known.

1.6 Abbreviations

CDSS Clinical Decision Support System

CIM Computer-Interpretable Model (of a CGP)

CGP Clinical Guideline or Protocol

DSS Decision Support System

EPR Electronic Patient Record

Chapter 2

Related Work

In this chapter, related work in the field of my thesis is divided three groups: Computer-interpretable guideline models (Section 2.1), temporal data abstraction (Section 2.2), and neighbouring field, which do not form part of the work presented here, but influence it to a noteworthy extent (Section 2.3.

2.1 Computer-interpretable models of guidelines and protocols

Section 1.1.1 established the benefits of clinical guidelines. One of the reasons for the lack of impact of guidelines lies in the fact that they are not available at the point of care or difficult to handle [25].

Embedding them in the care process facilitates the spreading of high standard practices that otherwise would have much less impact [62]. Modelling guidelines as CIMs and integrating them at the point of care was shown to increase adherence to the guideline [82, 61].

LIFe-reader®, a DSS implemented in a PDA with barcode reader, for use by nurses in homecare, was perceived as easily providing drug-related information on the patient (called patient profile in this context) and improve the quality of the medication [81].

In bigger projects, knowledge modelling issues become important. Many representations for CIMs have been proposed. Section 2.1.1 gives a brief overview of them, focusing on the topic of this thesis – temporal data abstraction and execution of the model. Section 2.1.2 summarizes comparison articles published by various authors. Ten Teije et al. [158] give a comprehensive introduction.

2.1.1 Individual guideline representations

All of the approaches discussed below feature interactive editing tools, which are beyond the scope of this thesis. In the following, I focus on a brief introduction, followed by a discussion of guideline execution and data abstraction facilities in each of the approaches. With the exception of Asbru, which form the basis of later chapters of this thesis, representation details are only described where they are characteristic or unique.

2.1.1.1 Asbru

Asbru was jointly developed by Silvia Miksch, Yuval Shahar, and Peter Johnson at Stanford in 1996 [95]. It is part of the Asgaard framework [135]. Retaining the original overall idea, its finer details were significantly revised in 2000, replacing the Lisp-style syntax by XML and defining precise syntax for the content of the knowledge roles [125]. This description is adapted from [127].

Asbru focuses on the representation of protocols, but guidelines can be modelled, too, as shown in the Protocure project (compare Section 5.1.2.1). Both are represented as time-oriented, skeletal plans. Skeletal plans are plan schemata at various levels of detail, which capture the essence of the procedure, but leave enough room for execution-time flexibility in the achievement of particular goals [55]. Thus, they are usually reusable in various contexts. Asbru expands the idea of skeletal plans through the addition of knowledge roles, a rich set of ordering of actions and plans, and temporal dimension of states, actions, and plans.

Time-oriented, skeletal plans are uniformly represented and organized in the plan library. Atomic plans - called actions or operators in the planning literature are modelled as user-performed plans. The language definition caters for external software objects as an alternative to user-performed plans, but practical implementations never used this feature. Instead, external wrappers bridge the Asbru interpreter to the embedding system.

Each non-atomic plan in the plan library is hierarchically composed of a set of plans with arguments and time annotations (representing the temporal scope of a plan). A plan is identified by its name and consists of five knowledge roles: preferences, intentions, conditions, effects, and a plan body (layout) which describes the steps to be taken. The major features of Asbru are that

- prescribed actions and states can be continuous;
- intentions, conditions, and world states are temporal patterns;
- uncertainty in both temporal scopes and parameters can flexibly be expressed by bounding intervals;
- particular conditions and operators are defined to monitor the plans execution; and
- explicit intentions and preferences can be stated for each plan separately.

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 constraints. Each parameter proposition contains a parameter name, a value description, a context description and a time annotation. The time annotation allows the representation of uncertainty in starting time, ending time, and duration. The time annotation supports multiple time lines (e.g., different zero-time points and time units) by providing arbitrary reference annotations. Temporal shifts from the reference annotation are used to define the uncertainty in starting time, ending time, and duration [42, 112]. To allow temporal repetitions, sets of cyclical time points and cyclical time annotations are defined.

In the case of plans which are not user-performed, the subplans defined in the plan body are performed either cyclical, sequentially, in parallel, unordered, or they are *any-order*-plans. While unordered subplans are completely free from any constraint, only one of a set of *any-order*-plans may be performed at a time, but their ordering is not fixed. Parallel plans start together, but no constraints on their ends are established. Sequential plans are executed exactly in the order in which they are given, without temporal overlap.

For each type of subplans (including unordered) none of the children are started before the parents start and all of them are terminated if the parent terminates. The success or failure of child plans are propagated to the parent under various schemes (defined by the propagation specification). The parent can have its own complete and/or abort condition and it can wait for the completion of all its children or of a subset of them (as defined in the continuation specification). Only if both the complete condition and the continuation specification are satisfied the parent terminates.

Both a plan itself and the activation of a plan can contain a time annotation limiting its temporal extend in the same way as it is done for parameters in the parameter proposition.

Section 4.5.1 discusses technical details of the language elements central to monitoring (time annotation, parameter propositions, and conditions). Section 4.6.1 contains further details on the plan semantics.

Apart from the Spock system described in Section 2.1.1.3, there are two implementations of Asbru execution systems.

Bosse's interpreter [22] consists of a parser written in Prolog which translates Asbru Light [130] to CLIPS rules, and a CLIPS programme implementing the Asbru semantics. It did not include monitoring of time-annotations. Temporal data abstraction is restricted to measuring the change between two succeeding values, and mapping a numeric value to a qualitative one. One of its fundamental problems was that it ran at constant speed. This speed could be increased by a given factor, but since this reduced the time available for data input, there was a limit to such speed-up. All data was entered interactively when prompted. The output was an execution trace in free-text. This tool was used to validate Asbru guidelines in the Protocure I project (compare Section 5.1.2.1), and to gain an insight into the Asbru semantics in the course of training Asbru modelling.

Fuchsberger reduced Asbru Light further, focussing on those language elements which were present in the guideline for artificial ventilation which he modelled. He implemented a new interpreter, AsbruRTM [56] in Java, based on the ideas in [93]. Temporal data abstraction was again omitted. Patient data was acquired from a dedicated user interface via CORBA.

2.1.1.2 Arden Syntax

Arden Syntax is the oldest approach, with its first version dating back to 1989 [33]. And it has since become an ANSI standard [1]. Knowledge is modelled in Medical Logic Modules (MLMs) [76]. Besides Maintenance and Library slots, holding administrative information, the Knowledge slot contains sub-slots for data, evoke, logic and action, among others. Data specifies the data used in the MLM and how it is retrieved

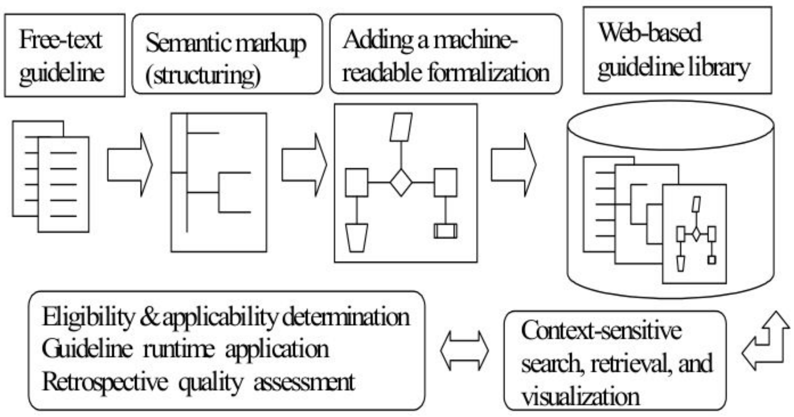

Free-text guideline
Semantic markup (structuring)
Adding a machine-readable formalization
Web-based guideline library
Eligibility & applicability determination
Guideline runtime application
Retrospective quality assessment
Context-sensitive search, retrieval, and visualization

Figure 2.1: The DeGeL approach to gradual guideline formalisation [136].

(in term specific to the local installation. Evoke describes the event triggering the evaluation of this MLM. Logic specifies the condition for carrying out the action defined in slot action.

Unfortunately, the content of the logic slot is not standardised. Consequently, each installation uses its own syntax for this key part which severely limits compatibility of models produced with tools from different vendors. Also referencing the data is not standardised.

Sherman et al. [140] proposed a complementary approach to better represent temporally complex plans and improve dealing with unavailable data. They store intermediate patient states in a central data repository. Like the syntax definitions described below, this solution is not part of the language specification.

Arden/J introduces mapper elements to improve the portability of MLMs [84]. McCauley et al. [91] used the language MUMPS (Massachusetts General Hospital Utility Multi-Programming System) [41, 171] to replace free-text in the Arden syntax. Kuhn et al. [87] use a C++-based precompiler for the same purpose. Tiffe [163] proposed an extension to the syntax of logics slot to incorporate fuzzy linguistic terms.

2.1.1.3 Digital Electronic Guideline Library (DeGeL)

The DeGeL framework [136] features a set of distributed tools (compare Figure 2.1). The focus is on the gradual conversion of guidelines from free-text to formal notations, via semi-formal representations. Formal target languages are Asbru (compare Section 2.1.1.1) and GLIF (compare Section 2.1.1.8). It supports mixed – or hybrid – representations, in which parts of the guideline are specified in a more formal way while others remain free-text.

Within the DeGeL framework, the guideline execution is implemented in the Spock execution engine [180]. It is able to process hybrid guideline models, e.g., hybrid Asbru [137]. The Spock system also adapts for the availability or lack of patient data. Missing data is queried from the user. If the guideline is fully formalised, and the required data is found in the electronic patient record, the execution of the guideline is automatic. Otherwise, interaction of the user is required. Published descriptions of

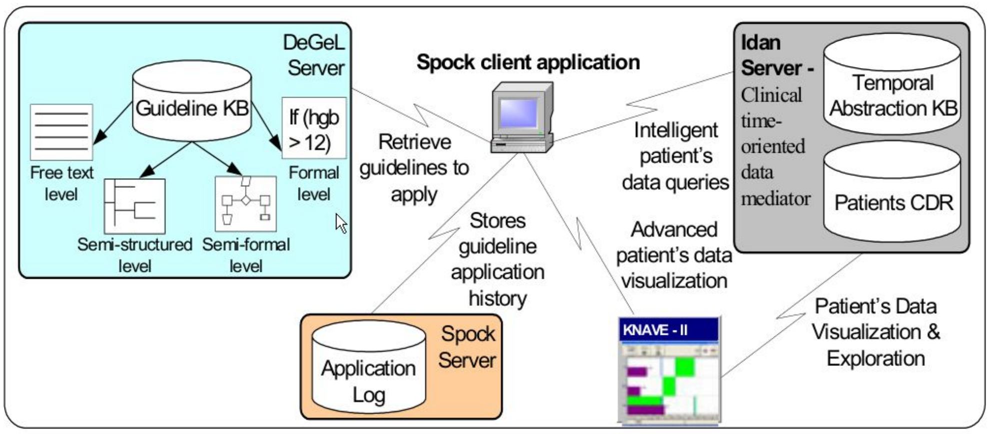

DeGeL Server
Guideline KB
Free text level
Semi-structured level
Semi-formal level
If (hgb > 12)
Formal level
Spock client application
Retrieve guidelines to apply
Stores guideline application history
Intelligent patient's data queries
Advanced patient's data visualization
Idan Server - Clinical time-oriented data mediator
Temporal Abstraction KB
Patients CDR
Patient's Data Visualization & Exploration
Application Log
Spock Server
KNAVE - II

Figure 2.2: The Spock architecture [179].

practical evaluation focus on the later case [181].

Figure 2.2 illustrates the Spock engines and the other modules related to guideline execution. The IDAN server supplies temporal abstractions, which are discussed in Section 2.2. KNAVE II [134] is a further development of the graphical user interface also used in the EON project, which is described next.

2.1.1.4 EON

EON was developed by the Stanford University. It is a framework for modelling and executing guidelines. It includes different servers to support specific tasks (compare Figure 2.3). The Padda guideline execution server [167], combines clinical guidelines formalised using the Dharma guideline ontology [165] and patient data to situation-specific recommendations.

The Dharma guideline ontology uses three different query languages: object-oriented, logic-based and temporal. The first provides form-based templates for simple cases. The second, Protege-2000s PAL language, implements a subset of first-order logic. The third, Tzolkin queries, are sent to the second server in the framework, the Tzolkin temporal data mediator, which is described in Section 2.2.3.5.

The WOZ server [139], complements the previous two by providing explanations. It features a model of argumentation based on the work of Toulmin [164], explicitly modelling data (what is given), warrant and backing of the warrant, claim (the result of the conclusion), qualifier (modulating the claim), and rebuttal (defining exceptions from the rule). Figure 2.4 shows their relation. The data, i.e., the dynamic input for triggering the argumentation process, comes from the graphical user interface called KNAVE [133].

Patient data is modelled as static (always valid), time-stamped, or associated with a time interval during which it is valid [168].

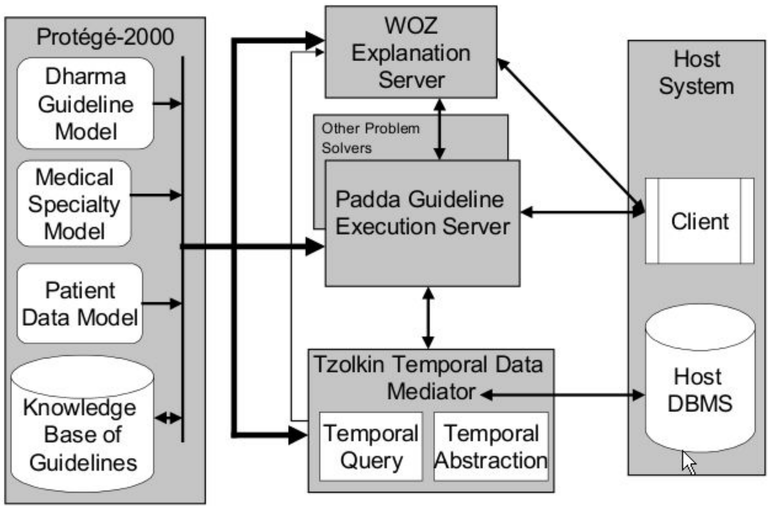


Figure 2.3: System architecture of EON [168].

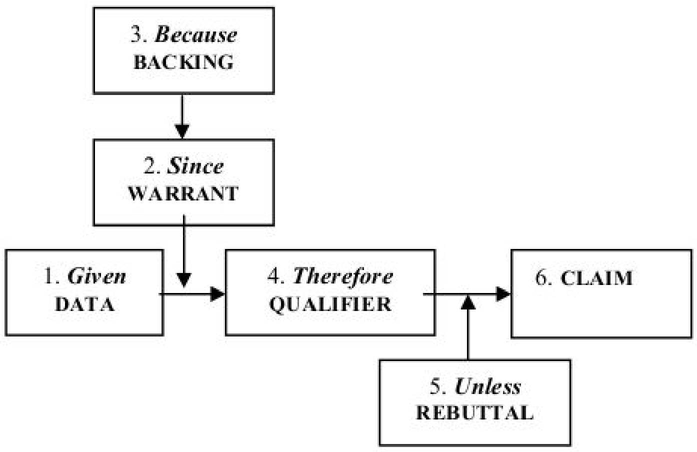


Figure 2.4: Toulmins argument structure as used in WOZ [139].

2.1.1.5 Guideline Elements Markup (GEM)

GEM was introduced by Shiffman et al. in 2000 [141]. In 2006, its second version (GEM II) became an ASTM standard [2]. Figure 2.5 shows the concept hierarchy. It focuses on high-level annotations of guidelines. For the clinical algorithm, it uses GLIF (compare 2.1.1.8) [142].

An execution engine was proposed which consisted of eXstensible Server Pages (XSP) transforming a GEM-annotated guideline into a web form into which the user would fill in all patient data [59]. It does not discuss storage and retrieval of data and does not appear to be developed further. Most work of this group deals with modelling and guideline appraisal.

2.1.1.6 GASTON

The GASTON framework consists of a guideline representation based on Problem Solving Methods (PSMs), an authoring environment, and an execution engine [36]. The latter consists of a core which is complemented by plug-ins to interface to patient record systems and monitors.

Four applications of the system are mentioned in [39], of which three concern low-frequency domains: GRIF: a reminder system that provides automated feedback on test ordering in general practice [18]; M-PADS: a psychopharmacological advisory system that provides decision support concerning the selection of the most suitable psychoactive drug [78]; and a system for the management of chronic diseases [37].

The application in the high-frequency domain is CritICIS. It is described as a real-time critiquing system used in critical care environments such as Intensive Care Units [35]. However, temporal data abstraction is not performed. The system focus is on checking manually entered data for completeness and on potentially undesired interactions between ordered drugs.

This is the only system discussed here which explicitly covers requirements of high-frequency domains, such as short response time to input, and tasks are described as event-based.

2.1.1.7 GLARE

The GLARE framework builds on ontology modelling work started in 1997 [162]. Figure 2.6 shows the ontological hierarchy.

Originally, the need for explicit primitives to represent conditions was denied [63]. Later, it integrated temporal reasoning for consistency checks within the guideline [161]. Similar to Asbru, plans (called actions here) have states active, suspended, aborted, and completed [160]. Decisions are modelled as separate decision actions. Data input is modelled as query actions. Treatment actions are named work action. Conclusion actions simply add a given name of a diagnostic hypothesis to a list recorded for the patient. They are omitted in later papers (e.g., [13]).

The system has been integrated using Java and an Oracle database. During execution, the steps already performed are marked in the task graph to show the user the current state of the treatment. Quoted sample applications belong to the low-frequency domain. Temporal data abstraction is not included [160].

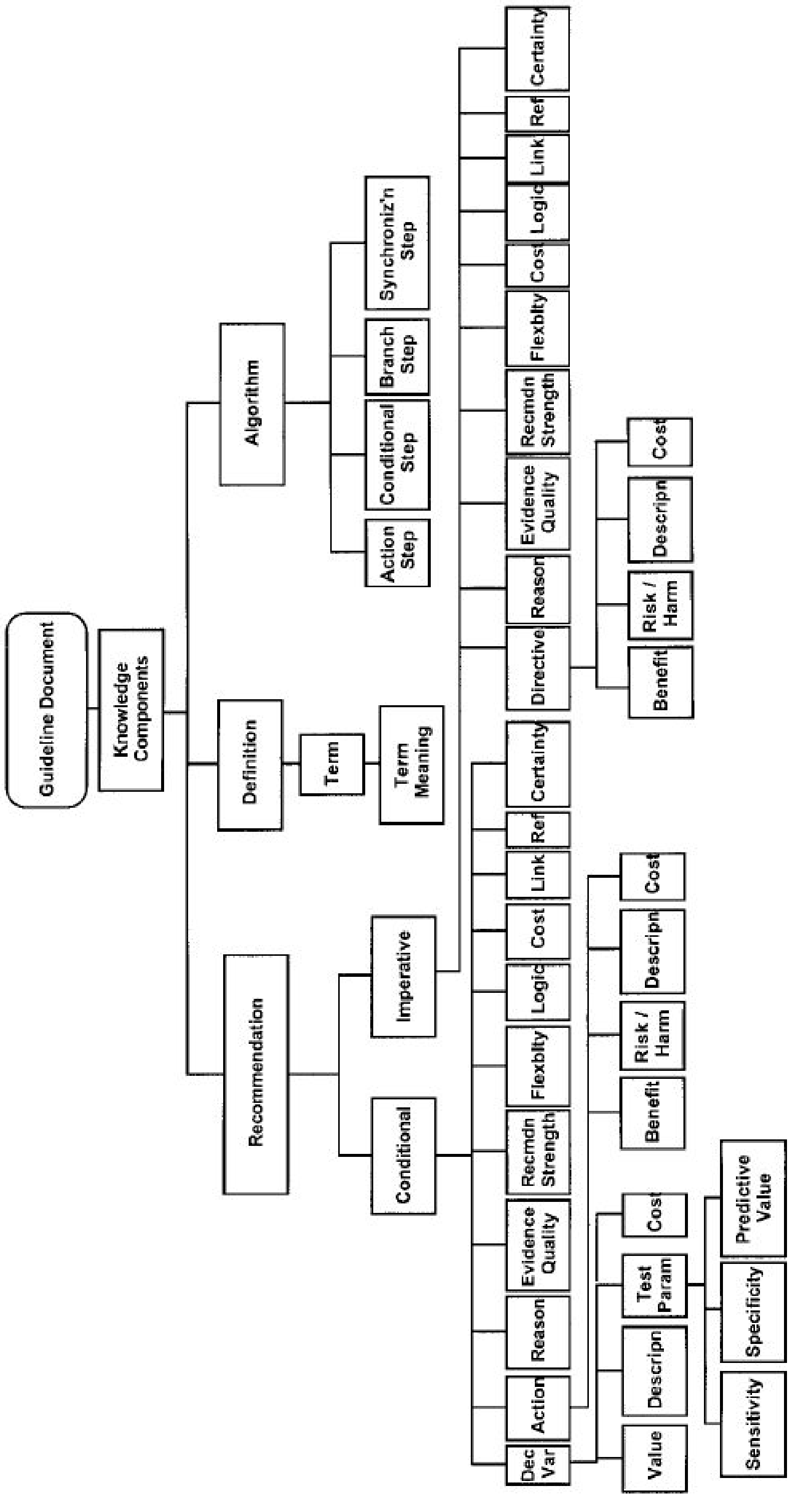


Figure 2.5: GEM concept hierarchy [141].

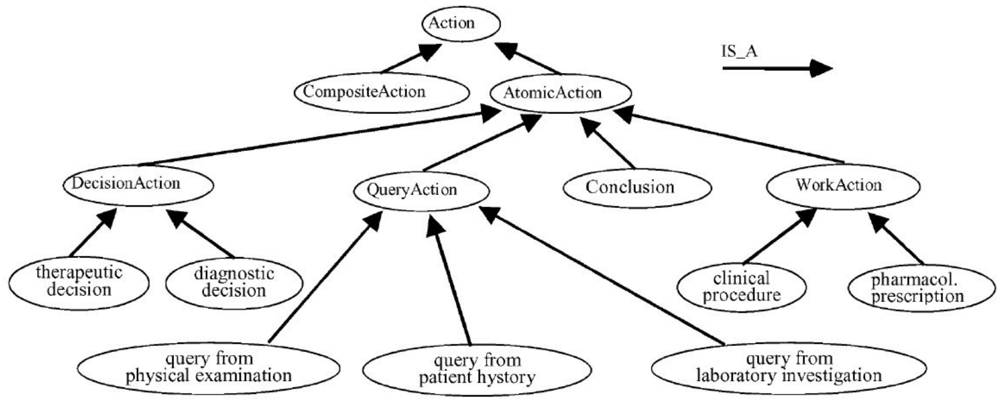


Figure 2.6: GLARE concept hierarchy [161].

A later paper [13] shows how GLARE models can be mapped to Petri nets [107]. In particular, they use stochastic well-formed coloured Petri nets [28] and a composition operator [16] to superimpose transitions or places with matching labels. However, this work does not seem to concern guideline execution in practice, but rather aim at presenting formal semantics for the GLARE representation similar to those defined for Asbru [11].

2.1.1.8 Guideline Interchange Format (GLIF)

The Guideline Interchange Format (GLIF) was developed by the InterMed collaboratory, a joint project of Harvard, Columbia, and Stanford universities [102]. It is based on four precursors: (1) the Arden Syntax (compare Section 2.1.1.2); (2) GEODE-CM, combining guidelines with structured data entry and data retrieval, in use at the Brigham and Women's Hospital; (3) MBTA, an architecture for building large knowledge-based medical systems, used at the Massachusetts General Hospital; and (4) EON a component-based architecture for guideline-based care and decision support, developed at Stanford university (compare Section 2.1.1.4).

Figure 2.7 shows the concept hierarchy. Care actions are formalised in action steps, conditional steps, branch steps, and synchronisations steps. Where Asbru has parallel and unordered subplans, GLIF has branch steps to start parallel actions and synchronisation steps to merge the parallel branches into one again.

A special construct is the k_of_n Criterion. It is fulfilled if k out of n criteria are fulfilled. The criterion elements in GLIF specify the eligibility criteria, resembling the filter-precondition in Asbru.

The original GLIF notation left room for further development as far as the precise representation of medical concepts, criterion logic, temporal information, and uncertainty is concerned.

The current version, GLIF3, features the use of Health Level 7 (HL7) standards and the ability to incorporate standard medical terminologies, features to manage complexity of large guidelines, a layered model to interface the patient data and medical knowledge involved, and the object oriented query language GELLO [23]. Its syntax is based on the Resource Description Framework (RDF) [88].

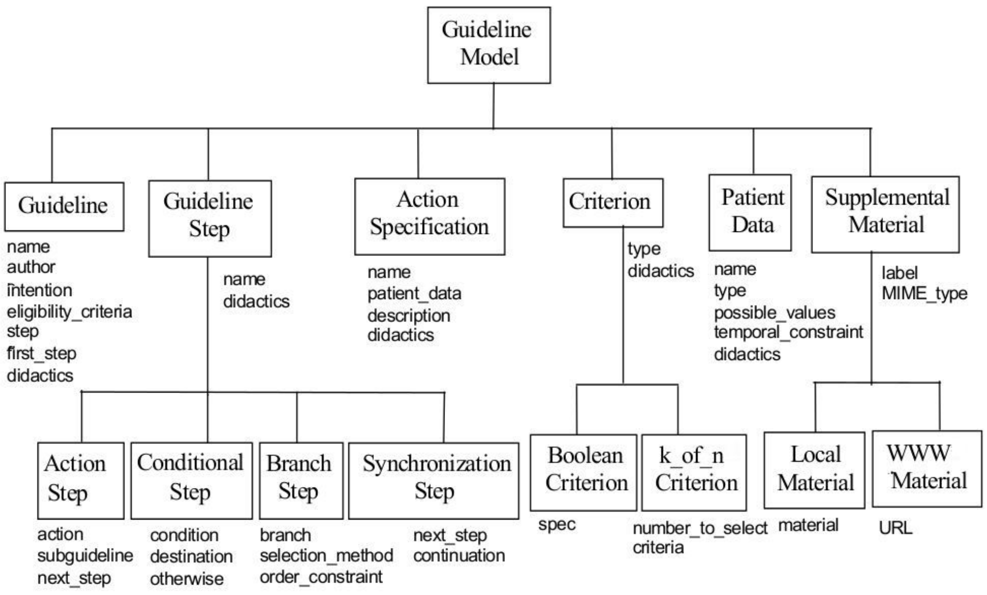


Figure 2.7: GLIF concept hierarchy, adapted from [102].

GELLO [152] is a further development of the Guideline Expression Language (GEL) [101], which was based on Arden Syntax. GELLO is based on the Object Constraint Language (OCL) [7].

GLEE, the Guideline Execution Engine for GLIF3 is designed as middleware to be integrated with the clinical information system at a local institution. It generates suggestions for actions to be taken by the user. If connected to a patient monitor, it produces these suggestions automatically. However, the authors stress the "system suggests, user controls" approach [173].

Besides clinical decision support, GLEE supports quality assurance, guideline development, and medical education. Combined with GELLO, it can specify concise conditions. However, GELLO is an object-oriented query language rather than a time-oriented one [174]. With EON being part of the InterMed collaboratory, the availability of the resources described in Section 2.1.1.4 seems to be assumed.

The Guideline Execution by Semantic Decomposition of Representation (GESDOR) model [172] bridges ProForma to GLIF by defining a modified version of the ProForma representation (with modified expression language, cyclic task execution, and patient data definition). An implementation of the GESDOR execution engine was shown to perform exactly like GLEE in test runs on two guidelines.

The "Online Guideline Assist" (OLGA) project[1] translated guidelines formalised in GLIF into workflow specifications formalised in JPDL (jBPM Process Definition Language[2]). Using custom middleware, the workflow execution was integrated into the patient data management system in use at the point of care, avoiding any additional

[1]`http://www.fit.fraunhofer.de/projects/prozesse/olga.html`, last accessed Oct 20th, 2010

[2]`http://docs.jboss.org/jbpm/v3/userguide/jpdl.html`, last accessed Oct 20th, 2010

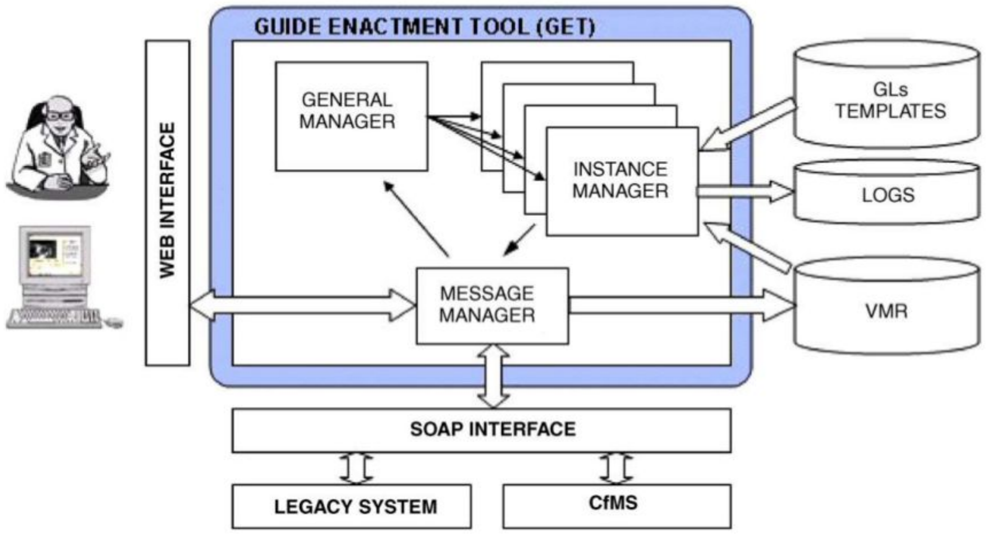


Figure 2.8: Guide Enactment Tool system architecture [31].

effort for the user and writing recommendations back into the patient record [120, 119, 118].

This system is evaluated in the field of weaning from long-term artificial ventilation in adult intensive care units. However, only the chain of checks and decisions is modelled. The latest readings of some values which could arrive at a higher frequency are accessed, but only once per day. Temporal data abstraction is not mentioned.

2.1.1.9 Guide and NewGuide

Developed at the University of Pavia, both representations share a flowchart-like representation of treatment and diagnosis steps, inspired by Petri Nets [107].

Guideline execution is performed by the Guide Enactment Tool (GET) in an interactive way. The GET uses a Workflow Management System to implement functions like waiting for a given period of time. Figure 2.8 shows the system architecture. Guide permits three levels of integration: (1) stand-alone system, with all data entered interactively, (2) data retrieved from the EPR at the point of care but user interface still supplied by the Guide system, and (3) interaction with Guide via the hospital information system [31].

The NewGuide system adds medical knowledge components, improved expressiveness of the external temporal server (see below), and GEM attributes to describe the guideline. Also, the graphical editor was improved. NewGuide models can be translated to Petri net models for simulation purposes [109].

Fields of application are pressure ulcer prevention, stroke (all phases from emergency to rehabilitation), and heart failure management. In the latter case, Guide is fully integrated with the existing EPR and careflow management system and manages the prescription of laboratory examinations by general practitioners [30].

The NewGuide system employs an HTTP-based server for temporal abstractions, which is capable of detecting qualitative trends and Allen relations [29]. However, for each abstraction to perform, the complete set of data must be sent to the server, together with the description of the abstraction, which then returns the abstraction [15].

2.1.1.10 HELEN

The HELEN-Project was established to introduce guideline-based care at the Department of Neonatology of the Heidelberg University Medical Center.

The Guideline Execution Engine serves to guide users through the various steps of guidelines. It is capable of sending messages to Personal Digital Assistants, but it does not process high-frequency data. Temporal data abstraction is not mentioned [145].

Two guidelines were implemented for an in-depths evaluation of the system: One for the management of hyperbilirubinemia in the healthy newborn as published by the American Academy of Pediatrics. The second deals with the management of apnoea in pre-term newborns. Management actions foreseen in this guideline range from food adaptation to introduction of mechanical ventilation, which itself is not controlled by the HELEN system. I.e., both applications belong to the low-frequency domain.

2.1.1.11 PLAN

This approach differs from most others in using Event-Condition-Action (ECA) rules [58, 104] to model clinical test ordering protocols [79].

The rules could easily become unmanageable in number for a realistically sized protocol. The PLAN language prevents this, providing a more conventional presentation of a protocol model and all its associated rules to the user [175].

This work builds on TRiPS (Test Request Order System) developed by the TUDOR project which also supplied an editor to conveniently edit the ECA rules from which a protocol is built [17].

Later, PLAN was further developed into an XML-based language, called AIM. The corresponding modelling and execution framework is called CIM (Complex Information Management framework) [90], SEM (Specification, Execution, and Management framework) [176] and SpEM (Specification, Execution, and Management) [44].

Although this approach is event based, it does not aim at high-frequency applications, but test ordering. Therefore, the authors do not give details concerning the response time of the system. Data abstraction is not within their scope either.

2.1.1.12 PRODIGY

Funded by the UK Department of Health and co-ordinated by the Sowerby Centre for Health Informatics at the University of Newcastle (SCHIN), PRODIGY (Prescribing Rationally with Decision-Support in General Practice Study) tried to bring decision-support system for drug prescription to clinical practice.

It can be seen as a forerunner of the other systems described here, in terms of practical application. In phase two, nationwide evaluation in general practices in the UK, implementations of decision support systems which were based on the ideas in the PRODIGY project achieved up to 27% use rate, which was a significant progress over phase one [113].

However, the system did not succeed to change guideline compliance which was attributed to the difficulty to show changes in an application domain such as chronic

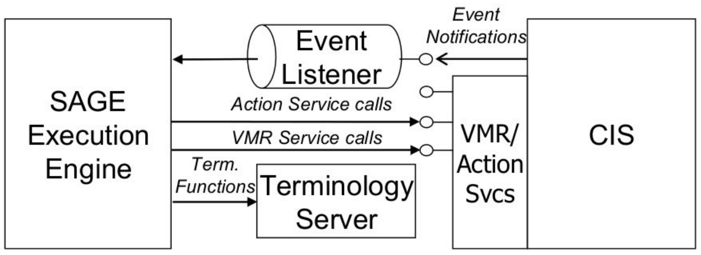


Figure 2.9: SAGE system architecture [110].
Abbreviations used in the figure:
CIS = clinical information system, VMR = virtual medical record,
Svcs = services, Term = Terminology.

diseases [49]. This was also connected to the premature state of the software [108]. Later studies confirmed these assumptions [57, 85, 144].

Temporal data abstraction was not involved [150].

2.1.1.13 ProForma

ProForma [155] was developed by Cancer Research UK. Its first version is commercialized under the name Arezzo^{TM}, a second version is implemented in the Tallis system [156]. It features an elaborate decision model where strong and weak arguments for and against an action can stated independently and the execution module weighs them against each other. This sets it off against other approaches where such decision processes are modelled explicitly.

The Arezzo guideline enactment engine guides the user through the collection of data. Emphasis is put on leaving all decisions to the end user [6]. Application is thus limited to low-frequency domains. Temporal data abstraction is not mentioned. In the low-frequency domain, the list of applications ranges from drug prescription at general practitioners to cancer referral.

Proforma is also used by the agent-based Health Care Services release 2 platform (HeCaSe2) [80].

2.1.1.14 SAGE

SAGE, the Standards-Based Active Guideline Environment project, can be seen as successor of the EON project. It builds on existing approaches, bridging them to workflow, standards, and electronic patient records [166]. The main focus is on the integration with the native clinical information system (CIS) and its user interfaces.

Figure 2.9 shows the system architecture. For maximum portability, the event listener is implemented as a web service and the SAGE project aims at interfacing the VMR/Action services with standards such as HL7. As a demonstrator, the SAGE system was integrated with IDX System Corporation's commercial Carecast^{TM} CIS. The demonstrated application population-based reminder scenario in the field of vaccination [110].

2.1.1.15 Representations without execution engine

The following projects developed representations or tools, but did not deal with guideline execution or temporal data abstraction.

HGML. HGML (Hypertext Guideline Markup Language) is an XML-based format to mark up conditions and recommendations within free text guidelines [65].

Prestige. The EU project Prestige is an early attempt to tackle computer-aided guideline-based care involving a large number of scientific and commercial partners. Already in 1998, it pointed out that the two major bottlenecks were the creation of formal models and the bridging to the EPRs [60].

Stepper. Stepper is an editor for document-centric guideline modelling in a multi-step process. Target representations can be Java or Asbru [116, 157].

2.1.1.16 Guideline execution without new representation language

The following projects deal with guideline execution but did not develop their own guideline representation language. Some of them were pioneering in bringing guideline-based care into care practice. However, within the community described above, it is established that modelling complex medical knowledge requires a dedicated representation. The motivation is to ease the knowledge engineering task rather than formal necessity.

ASTI. The ASTI (Aide à la Stratégie Thérapeutique Informatisée) project uses if-then rules, decision trees and a drug database to improve the practice of drug prescription. It has two modes of operation. In the critic mode, the orders entered by the user are evaluated against this knowledge base and constructive criticism is displayed as appropriate. In the guided mode, the user is patient-centred solution to a given therapeutic problem [123]. It is deployed in the management of arterial hypertension [122].

ONCOCIN. Developed for the Stanford Oncology Day Care Center since 1979, ONCOCIN is one the founding projects in the field. It is a consultation system for chemotherapy dose selection and included a customised keyboard with 21 extra keys to enter the required information with minimal typing. It was one of the first to emphasise the need for symbolic representation of knowledge and to explain the reasoning behind a recommendation [117]. It was also one of the first to show that use of a computer-supported DSS improves the quality of data entry [86], and that modelling the patient history and representing temporal relations and facts is a important part of the challenge [83].

OncoDoc. OncoDoc is a decision support system based on decision trees and hypertext. It is focussed on treatment selection in oncology. It is used at Institut Gustave

Roussy to increase physician awareness of trials open to new participants and lead to an increase of patient enrolment in clinical trials by 17% [121].

In an ongoing clinical trial evaluates the effect of the routine use of OncoDoc2 during multidisciplinary staff meetings regarding the compliance of decisions made in these meetings with guidelines [5].

PRESGUID. The PRESGUID project implemented guidelines in the form of decision trees linked to a commercial drug database and recommendations in XML format [46]. In a comparison between physicians using paper-based guidelines and the PRESGUID system in hypertension management, diabetes mellitus treatment and peripheral arterial disease, the group using PRESGUID showed significantly higher compliance rates [45].

2.1.2 Comparisons

Over the years, a series of authors compared approaches to guideline modelling and execution. Their work is summarised in this subsection.

2.1.2.1 Peleg et al. 2003

An international consortium modelled sections from two guidelines in Asbru, EON, GLIF, GUIDE, PRODIGY, and ProForma. Both guidelines dealt with low-frequency domains, namely managing chronic cough and prevention, detection, evaluation, and treatment of high blood pressure. The study focussed on the representations and their individual features, not on guideline execution. It comprised eight dimensions:

1. organization of guideline plans,
2. goals,
3. model of guideline actions,
4. decision model,
5. expression language,
6. data interpretation/abstractions,
7. medical concept model, and
8. patient information model [106].

The comparison of language features showed that Asbru only lacked regarding the expression of preferences associated with plans and in argumentation rules. Here, ProForma offers a complete framework while other representations (including Asbru) imply the preferences in the conditions modelled. Decision trees were found to be missing but the *if-then-else* element covers this functionality (in more recent language versions).

2.1.2.2 De Clercq et al. 2004

The authors compare acquisition, verification, and execution aspects for the following representations: Arden Syntax, GLIF, ProForma, Asbru 6.3 and EON [34]. An updated version has been published in 2008 [34].

Regarding temporal reasoning, they state that "the Asbru approach contains the most sophisticated structures. EON and GLIF both adopt a subset of the Asbru temporal language."[34, p. 38]

2.1.2.3 Isern & Moreno 2008

The authors compared guideline representations and architectures for creating and executing them for the following execution engines. The name of the representation is stated in brackets where it has its own name: Arezzo$^{\text{TM}}$ (ProForma), DeGeL (Hybrid-Asbru), GLARE, GLEE (GLIF3), HeCaSe2 (ProForma), NewGuide, SAGE, SpEM (PLAN) [79].

They found the following two common limitations in these approaches:

1. While the representations are rather similar in their features, there is no standardization. Each execution engine uses its own representation.
2. The problem of mapping to the EMR is not solved in a general way. Instead, individual solutions are created in each case.

They also point out that health care is not fully computerized in most countries, and therefore, it is not astonishing that – except for some limited use of Arezzo$^{\text{TM}}$ – these systems have not been used in daily routine.

Execution engines are divided into two groups: event-based and rule-based. In the first case, events are handled by the system as they appear. This group comprises SpEM, GLEE, HeCaSe2 and SAGE. In the second case, rule execution is triggered by the user. This group comprises NewGuide, GLARE, DeGeL and Arezzo$^{\text{TM}}$.

For high-frequency domains, the event-based paradigm is the appropriate one, since alarms must be generated and treatment must be adjusted as indicated by arriving data, independent of the time and inclination of the care personal. In contrast, for low-frequency domains, the rule-based approach is suitable, since data entry is sparse and it occurs when the care staff deals with a patient, e.g., at an encounter in the practice, and only then recommendations are appreciated.

2.1.2.4 Weng et al. 2010

This paper focuses on the formal representation of eligibility criteria. For 27 different approaches, it discusses expression language, representation of patient data and representation of medical concepts. Only the first is relevant for this thesis. In this regards, the authors state that "Asbru excels in expressing temporal constraints among events." [174, p. 456].

2.1.3 Discussion

Despite the large number of representations proposed, the differences in syntax do not map to equally significant differences in the basic capabilities of the system.

Name of syntax and execution engine	Temporal data abstraction	Data-driven implementation
Arden Syntax	not in core definition [76]	various implementations
EON	subset of Asbru [40]	yes [168]
Hybrid Asbru/DeGeL	yes [179]	no [79]
GASTON	no [35]	yes [35]
GEM	as GLIF [141]	no [59]
GLARE	no [160]	no [79]
GLIF3, GELLO/GLEE	subset of Asbru [40]	yes [79]
NewGuide	via external server [29]	no [79]
HELEN	no [145]	yes [145]
PRODIGY	no [150]	no [113]
ProForma/Arezzo^{TM}	no [155]	no [79]
ProForma/HeCaSe2	no [155]	yes [79]
SAGE	based on GELLO [166]	yes [79]
PLAN/SpEM	no [79]	yes [79]

Table 2.1: Comparison of executable CIMs regarding (1) the availability of temporal data abstraction within the system, and (2) whether progress is driven by data arriving without direct user intervention or not.

Most systems focus on application on low-frequency domains. As a consequence, their natural mode of application is interactive, with the user triggering individual guideline steps and/or entering patient data. In such a setting, computational effort in data abstraction is not an issue.

Table 2.1 shows the two aspects most relevant for this thesis, for all the CIG representations:

1. Are there temporal abstraction facilities, and how do they compare to Asbru?

2. Does execution of the CIG progress according to data arriving without user intervention, or is it driven by requests from the user to take the next step or perform the next decision.

The conclusion from the table is that there are several approaches featuring temporal data abstraction and many systems are data-driven. However, only the group formed by EON, GEM, GLIF, and SAGE features both temporal data abstraction and a data-driven implementation. For this group, the expressiveness of the data abstraction facilities can be considered a subgroup of Asbru 7.4 [40]. Therefore, Asbru is selected as the sample representation language, for which solutions are presented in this thesis.

Relation	Description	Inverse relation
A before B	A ends before B starts.	B after A
A during B	A starts after B starts and ends before B ends.	B contains A
A overlaps B	A ends after B starts.	B overlapped-by A
A meets B	A ends at the same time B starts.	B met-by A
A starts B	A and B start at the same time.	B started-by A
A finishes B	A and B end at the same time.	B finished-by A
A equals B	Both start and end of A and B are exactly the same.	The equality relation is reflexive.

Table 2.2: Temporal relations defined by Allen [8].

2.2 Temporal data abstraction

In this section, I first describe conceptual work as far as it is relevant for the later described systems, followed by implemented systems. Then, I discuss how these approaches relate to the challenges associated with guideline execution in high-frequency domains. A recent overview of temporal data abstraction in the medical domain is given in [38].

2.2.1 Conceptual work

2.2.1.1 Allen's temporal relations

Allen [8] defined 13 temporal relations defined by the relations of their start and end points. Six of them are the inverse of 6 others, as shown in Table 2.2. These definitions were used by many researchers, most of whom only modelled those 7 which are not the inverse of others. Also the definition of temporal relations in Asbru builds on them.

2.2.2 Definitions by Shoham

Shoham [143] tried to combine the work of McDermott and Allen, pointing out shortcomings in their work. He defined a set of relations between the truth value of propositions over different intervals

downward-hereditary. Whenever a proposition holds over an interval, it holds over all that intervals subintervals, possibly excluding its end points.

upward-hereditary. Whenever a proposition holds for all proper subintervals of some non-point interval, except possibly at that intervals end points, it holds over the non-point interval itself.

gestalt. A proposition never holds over two intervals, one of which properly contains the other.

concatenable. Whenever a proposition holds over two consecutive intervals, it holds also over their union.

solid. A proposition never holds over two properly overlapping intervals.

These definitions were used in the RÉSUMÉ system discussed in Section 2.2.3.3.

2.2.2.1 TSQL2

TSQL2 merged the various approaches into a suggestion for an extension of SQL. Its features are: snapshot equivalence and identity are synonymous; support for only one valid-time dimension; tuple timestamping is employed; it is based on homogeneous tuples. valid time support for both the past and the future; timestamp values are not limited in range or precision [146, 148].

After some modifications and defining formal semantics for TSQL2 [21], it was approved by the ANSI SQL3 committee and forwarded to the ISO SQL3 committee [149]. However, due to differences, the project was cancelled [68, 147].

None of the existing database implementations fully supports TSQL2, but the Oracle Database 11g Workspace Manager supports a range of its features [12]. In addition, a range of database products support features described in TSQL2.

2.2.3 Implemented systems

2.2.3.1 TrenDx

Haimowitz et al. [67] developed TrenDx. It models temporal pattern in trend templates. These patterns consist of a partially ordered set of temporal intervals with uncertain end points. Value constraints – which are polynomials of degree 0, 1, and 2 to specify constant values, linear and quadratic growth or decrease – are linked to each of the temporal intervals.

Matching can be regression-based or constraint-based. The second approach is computationally attractive. The first is more robust to noise at a higher computational cost. It also provides a statistically grounded ranking of competing patterns, based on the mean absolute percentage error of each hypothesis. However, to reduce the otherwise prohibitive computational complexity of the search, the number of hypothesis under consideration is restricted. This number must be chosen carefully, since a too small value makes the algorithm susceptible to noisy signals.

Le [89] evaluated TrenDX in the field of endocrinology (paediatric growth monitoring) in a study involving 22 physicians rating 95 children and found that TrenDx performed worse than physicians. He attributes this finding to insufficient models (i.e., definitions of templates) and the inappropriateness of linear regression in situations with very scarce data, like in endocrinology.

In addition to endocrinology, Haimowitz et al. applied TrenDx in the field of artificial ventilation in intensive care, in a small experiment on a single patient [66].

2.2.3.2 VIE-VENT

VIE-VENT [92, 75] is a knowledge-based system written in CLIPS[3] to support physicians in the task of artificial ventilation in neonates. It features data abstraction, knowledge-based data validation and repair, selection of therapy recommendations, and visualisation.

[3]`http://clipsrules.sourceforge.net/WhatIsCLIPS.html`, last accepted October 20th, 2010

The data abstraction methods are the following: transformation of quantitative point data into qualitative values using context-specific schemata for data-point transformation; smoothing of these schemata for data-point transformation where the context changes; smoothing of data oscillating near limits between qualitative regions; context-sensitive adjustment of qualitative values (implemented by rules operating on qualitative values); transformation of interval data using schemata for trend-curve fitting. Data smoothing near thresholds is terminated when the input value exceeds the limit of region plus a predefined value, or after a predefined duration of constant smoothing application. Trend curve fitting uses a complex definition of the expected sequence of measurements when returning to region of the *normal* values, which is an important concept in artificial ventilation.

Figure 2.10 shows complex arrangement of functions for knowledge-based data validation and repair. They perform the following functions:

Range checking determines if a quantitative value is within a plausible range. This is performed for raw values and for trends.

Causal dependencies invalidates a second parameter if the first is invalid, as specified in a dedicated dependency rule set.

Functional dependencies specified for qualitative or quantitative values are used of error checking on the one hand, and data repair on the other hand.

Temporal validity determines the length of the interval starting at the measurement for which the value is valid (unless a new value is read or the value is explicitly declared invalid.

Stability check enforces a short time period during which a previously invalid parameter has to be stable before it becomes valid again.

Cross-validation checks for consistency of different parameters. It is typically used to validate a frequently available parameter against a more reliable, but less frequently available parameter which reflects the same measured entity, e.g., two sources of partial pressure of CO_2 in the blood.

Dynamic calibration repairs invalid values during a time interval. It applies a calibration function to a continuously-assessed parameter utilizing a valid (discontinuously-assessed) parameter, relying on the assumption that the invalid, continuously-assessed parameter has a valid trend.

Modified Hojstrup method. The original calculates a prediction for each new data point, based on the mean, variance and point-to-point correlation of the already known time series. If the difference is too big, the new data point is discarded [73]. The VIE-VENT version uses the deviation of the last two values from the mean instead of the correlation of the past measurements in the prediction calculation to obtain response times suitable for online processing of high-frequency data.

Functional trend dependency checks. Based on qualitative abstractions of trends, a set of rules define cases in which values are not plausible. They compare the

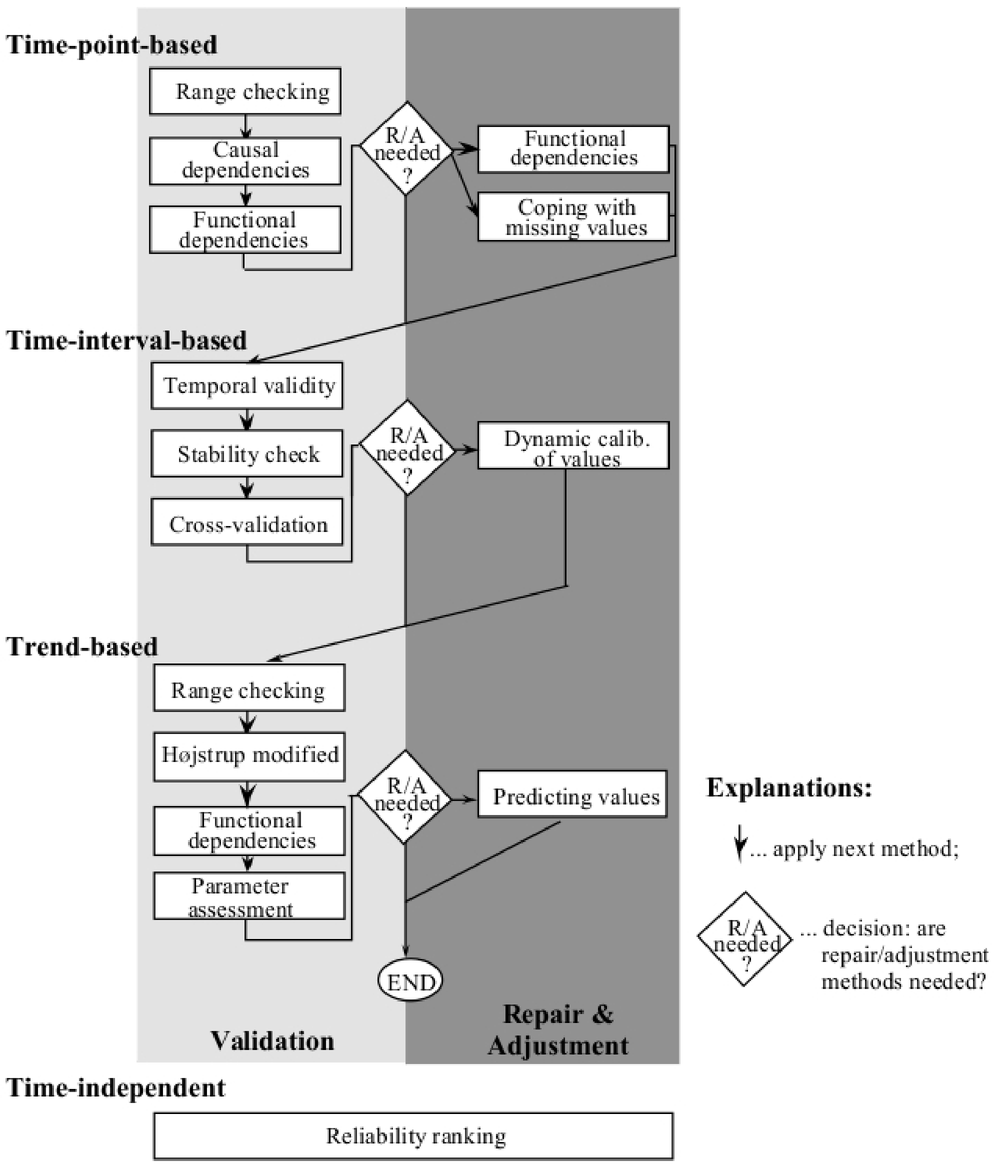


Figure 2.10: The VIEVENT modules for data validation (on the left) and for repair and adjustment (on the right) [92].

behaviour of two different parameters. If plausibility rules are violated, both values are marked ambiguous.

Parameter assessment for trends discards the last measurement if the trend is not in the same or a neighbouring qualitative region, i.e., if the change is too abrupt in qualitative terms.

Predicting values uses the last valid measurement while still valid and optionally the last trend. Both are controlled by plausibility checks on different levels to limit prediction to safe cases.

The generation of treatment recommendations is based on rules in a stepwise process as follows.

1. All recommendations matching the input are collected, maintaining the information on the basis of the recommendation.
2. Importance ranking of parameters gives priority to recommendations based on more reliable input sources. It also ranks reactions to decreased blood saturation higher than reactions to increased values.
3. The priority lists of attainable goals ranks recommendations such that certain options are tried first. Only if they fail twice, the next one on the priority list is tried.
4. Finally, any inconsistent recommendations left are removed from the list.

Figure 2.11 shows the user VIE-VENT interface.

Experiences collected with VIE-VENT lay the foundations for both the design of Asbru and the Pulsoximetry project described in Section 5.1.1.

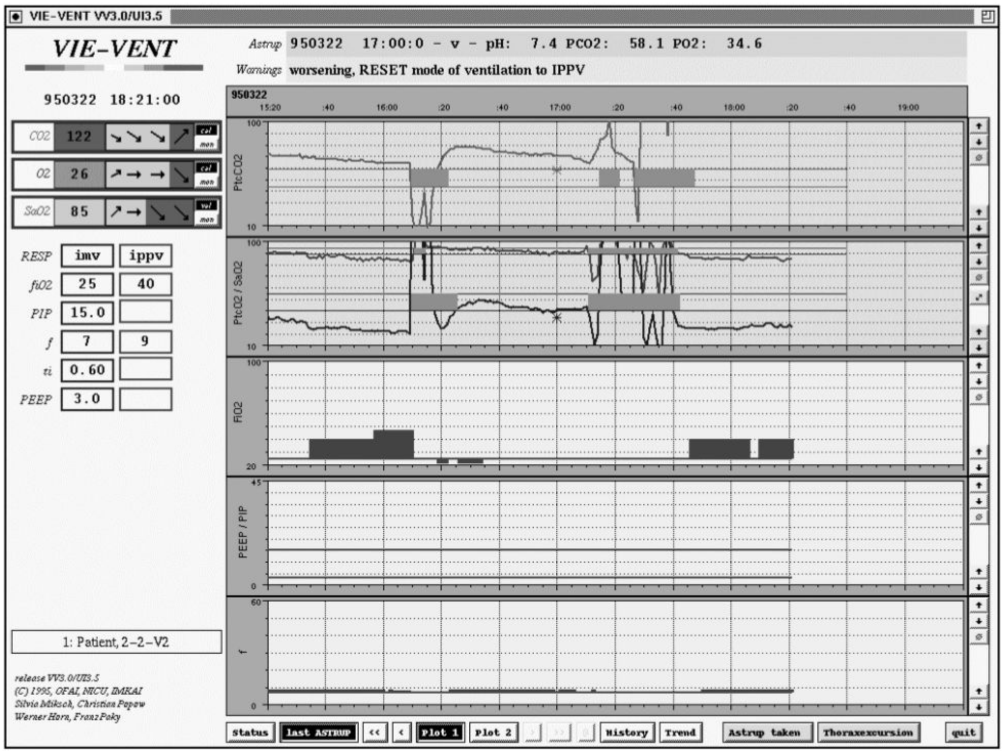
Belal et al. [14] further developed the ideas in VIE-VENT using fuzzy logic. In a clinical evaluation if their system (also in artificial ventilation of neonates), they found 91% agreement between actions suggested by the system and physician decisions.

2.2.3.3 RÉSUMÉ

RÉSUMÉ [132] distinguishes five subtasks of the abstraction task: temporal context restriction, vertical temporal inference, horizontal temporal inference, temporal interpolation, and temporal pattern matching. Each of them is supported by its own problem-solving mechanism (compare Figure 2.12).

Temporal Context Restriction. The context-forming mechanism transforms abstraction goal intervals, event intervals, abstractions and existing context intervals into context intervals based on the domain's temporal abstraction ontology.

Vertical Temporal Inference. The contemporaneous-abstraction mechanism transforms contemporaneous parameter points, parameter intervals, and context intervals into abstraction points and intervals of type state based on the parameter ontology.



Astrup 950322 17:00:0 - v - pH: 7.4 PCO2: 58.1 PO2: 34.6

Warnings worsening, RESET mode of ventilation to IPPV

950322 18:21:00

CO2	122	↘ ↘ ↘ ↗
O2	26	↗ → → ↘
SaO2	85	↗ → ↘ ↘

RESP	imv	ippv
fiO2	25	40
PIP	15.0	
f	7	9
ti	0.60	
PEEP	3.0	

1: Patient, 2–2–V2

release VV3.0/UI3.5
(C) 1995, OFAI, NICU, IMKAI
Silvia Miksch, Christian Popow
Werner Horn, Franz Paky

Figure 2.11: The user interface of VIE-VENT. To the top-left, qualitative temporal abstractions of blood gas measurements are shown. The actual and recommended ventilator settings are shown below. To the right, important parameters are plotted over the most recent four hours. Grey bars indicate intervals of data which was found to be invalid [92].

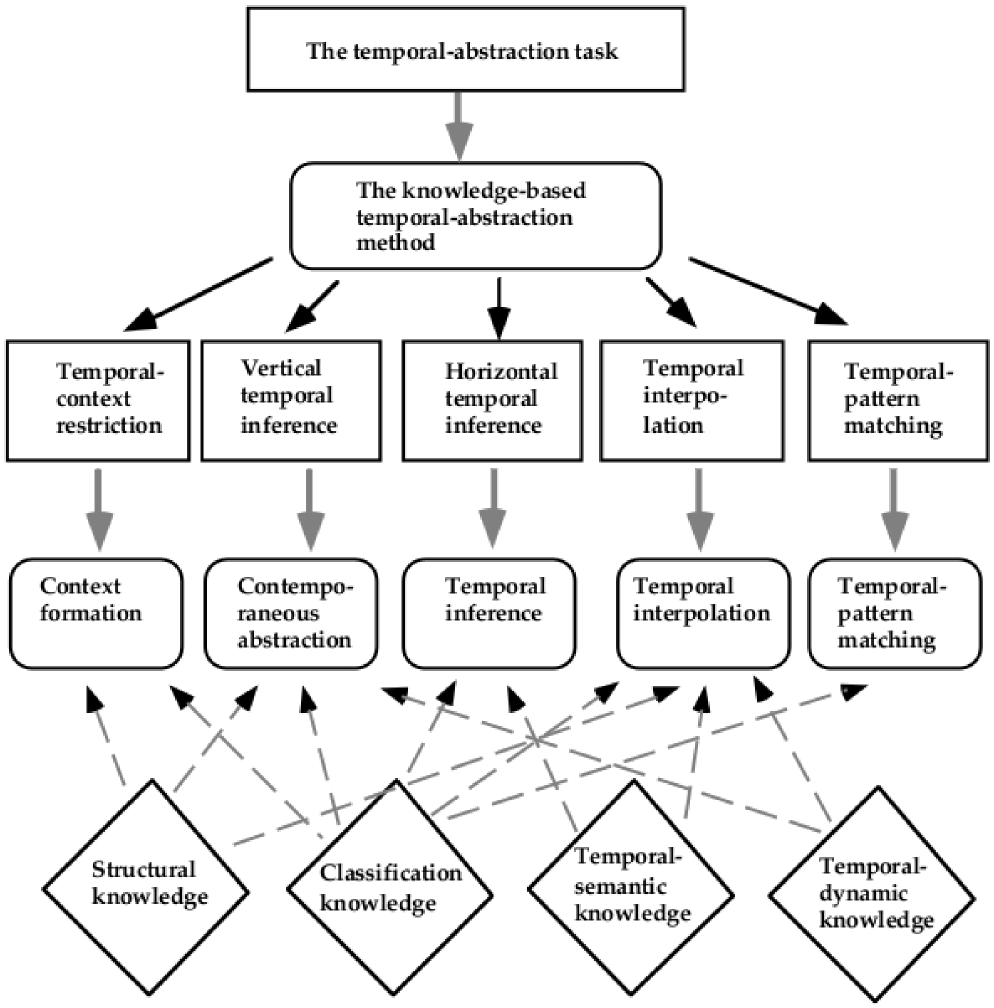

The temporal-abstraction task
The knowledge-based temporal-abstraction method
Temporal-context restriction
Vertical temporal inference
Horizontal temporal inference
Temporal interpo-lation
Temporal-pattern matching
Context formation
Contempo-raneous abstraction
Temporal inference
Temporal interpolation
Temporal-pattern matching
Structural knowledge
Classification knowledge
Temporal-semantic knowledge
Temporal-dynamic knowledge

Figure 2.12: The RÉSUMÉ temporal abstraction method [132]. The rectangles denote tasks; the rounded rectangles denote methods; the diamonds denote knowledge types.

Horizontal Temporal Inference. The temporal-inference mechanism performs two subtasks: temporal-semantic inference and temporal horizontal inference. The former infers specific types of interval-based logical conclusions, given interval-based propositions, using a deductive extension of Shohams's temporal semantic properties [143].

Temporal horizontal inference determines the domain value of an abstraction created from two (or more) joint abstractions, e.g., *moderately increased* and *significantly increased* can be joint to a new abstraction *increased.*

Temporal Interpolation. The temporal-interpretation mechanism bridges gaps between time points or time intervals, using local (forward and backward) and global (between two abstractions) truth-persistence functions. The output is an interval during which a parameter holds.

Temporal Pattern Matching. The temporal-pattern-matching mechanism extends the temporal-inference and temporal-interpolation mechanisms by abstracting over multiple intervals and parameters, and typically reflects heuristic domain- and task-specific knowledge.

The computational complexity is quotes as $O(N^3)$ per basic temporal abstraction mechanism, where N is the number of relevant parameter propositions, where a parameter propositions is a combination of a parameter, a parameter value, and an interpretation context.

A prototype of RÉSUMÉ was implemented in the CLIPS[4] expert system shell.

2.2.3.4 Chronus II

Chronus II is a temporal database mediator which extends SQL by temporal queries [100]. It supports valid time for entries in the database, Allen relations [8], temporal joins [98] and temporal indeterminacy [99]. Temporal information is stored in dedicated fields of standard SQL tables. Each request to Chronus results in a query for complete join of the relevant tables. From this potentially large result set, the Chronus system removes those which do not meet the temporal constraints. Temporal indeterminacy is represented as intervals within which a probability distribution function holds. E.g., "probably late" in a day is represented by increasing probability within the temporal interval constituting the day in question.

2.2.3.5 Tzolkin

Tzolkin complements Chronus and RÉSUMÉ by another mediator module [96]. It takes a query containing temporal abstractions and checks whether these abstractions have already been created and stored in the database. If this is not the case, RÉSUMÉ is invoked to produce them and store them in the database. Then, Chronos is invoked to query the database of abstractions and create the answer to the original request sent to the Tzolkin system.

[4]`http://clipsrules.sourceforge.net/WhatIsCLIPS.html`, last accessed October 20th, 2010

Tzolkin matches the probability rating returned by Chronus and only considers such solutions for which the probability rating exceeds a probability factor given as an argument of the query.

2.2.3.6 CAPSUL

The Constraint-based Pattern Specification Language CAPSUL expands RÉSUMÉ by the definition of periodic patterns. A periodic pattern is a series of intervals, during each of which exactly one instance of the repeating event occurs [26].

Gaps between intervals are defined by their minimum and maximum duration. These definitions must remain constant over all repetitions. Temporal relations between intervals are defined by Allen's 7 relations. Composite repeated patterns must retain their temporal relation through all repetitions.

Gap constraints are evaluated for pairs of intervals only once, which results in a computational complexity which is proportional to the number of intervals, for the verification of a single constraint, given a set of sorted elements. The constraints are designed specifically to maintain this linear effort, which sets the approach off from more theoretical work permitting exponential growth of combinations under consideration.

CAPSUL was experimentally used in oncology. It was found that the huge amount of abstractions generated by RÉSUMÉ was intractable for human expert who should evaluate them. In conclusion the strength of RÉSUMÉ lies in finding patterns involving multiple parameters on a higher abstraction level. A human expert could not easily find such patterns and the higher abstraction level reduced the data volume [27].

2.2.3.7 Momentum

Both Tzolkin and Chronus II suffer from computationally expensive, especially when population querying is involved. The Momentum system overcomes this by implementing temporal abstraction in event-condition-action rules. It dynamically generates and stores abstractions, similar to Tzolkin, but performed dynamically and incrementally. Unlike Tzolkin and Chronus-II, Momentum generates new abstract concepts only when relevant primitive (raw) data is added to the system, without recomputing previously generated abstractions [153].

2.2.3.8 IDAN

IDAN further develops the ideas in Tzolkin into a modular, distributed architecture incorporating the use of controlled vocabularies. Its default module for temporal reasoning tasks is called Alma [19]. It implements the expressiveness of CAPSUL [20] and uses the language TAR (for Temporal Abstraction Rules), which is a typed logic languages supporting subject types (e.g., patients), time (time points, intervals, and durations) and values [10].

2.2.3.9 KDOM

The Knowledge-Data Ontological Mapper (KDOM) provides basic temporal abstractions, but "is not intended to be a formal method for the specification of elaborate temporal-abstraction knowledge ... as is done in the RESUME and IDAN projects" [105, p. 187]. The main focus is on bridging guideline execution to patient records in practice.

2.2.3.10 Hunter et al.

Hunter and McIntosh [77] proposed an abstraction based on linear regression lines which were joined until they were too dissimilar. This resulted in long lines for steady data and short lines at points of change or noise.

Rules referring to these steady lines were applied, to detect patterns in groups of signals monitored in parallel. It was experimentally evaluated in an neonatal care unit.

2.2.4 Discussion

The aim of this thesis is to bridge temporal data abstraction to guideline execution. Asbru was chosen as the reference representation for guidelines based on the discussion given in Section 2.1.3. Therefore, this discussion focuses on two aspects:

- How to the above concepts map to Asbru?
- Which abstraction methods should be added to the original Asbru (version 6.4) definition to extend guideline execution by capable temporal data abstraction methods?

Asbru stores the history as a series of time-stamped measurements. However, this event-based approach is complemented by defining a trust period for each parameter, which expands the historic recordings to intervals during which given proposition holds.

Asbru includes those 7 Allen relations which are not the inverse of another one. The inverse relations are denoted by swapping the arguments.

Shoham's properties [143] apply to Asbru as follows. All parameters are *downward-hereditary* and *concatenable*. Parameter proposition resp. the intervals matching them are *gestalt*.

VIE-VENT contributes the following items to the list of desired functionality for the system described in this thesis:

- Transformation of quantitative data to qualitative values using context-specific schemata for data-point transformation. When the context changes, the limits between the qualitative regions change, too. This can lead to undesired changes in the qualitative output – a stable quantitative input maps to a different output after the context switch. However, it can also be argued that this difference is intended.

- Smoothing of oscillating quantitative input to produce stable qualitative output.

- Rules operating on quantitative and qualitative values, to adjust the abstractions in freely configurable ways.

- Trend curve fitting could be substituted by rules stating how long the transition from one qualitative value to the next better one is expected to take.

- Comparing qualitative and quantitative values of the measurements and their trend, for pairs of input channels, or with constants, forms the basis of a whole range of checks.
- Temporal validity is implemented in Asbru by specifying the *trust period* for each parameter.
- Stability checks call for user-defined delays in the abstraction network.
- Simple versions of the Hojstrup method can be constructed based on a regression model of a few most recent measurements.
- Predicting values is a delicate topic in medical care. If rules can be found which define the physician's strategy when faced with missing values, the model is guaranteed to meet the expectations, which is not as easy with simulation and prediction.

The abstraction tasks defined by RÉSUMÉ are implemented in Asbru as follows.

Temporal Context Restriction. The context is implemented as a set of qualitative variables defined by the designer of the plan library. The values of each context variable can be set independently by either a statement in a plan, or by the user. In addition, the value of a context variable can be derived from other values through data abstraction.

Vertical Temporal Inference. There are three forms of *classification* : (1) the abstraction of a qualitative parameter based on translation tables, or (2) based on logical expressions, and (3) the abstraction of Boolean values. These abstractions are context dependent – each translation table or each condition of such a parameter is valid for a certain context, which itself is defined as a logical expression based on context variables and their values. *Computational transformation* is performed through a set of arithmetic or logical combinations of parameters.

Horizontal Temporal Inference. Temporal-semantic inference of properties defined by Shoham is discussed above. Temporal horizontal inference is implemented in Asbru by combining arbitrary combinations of qualitative values into a new qualitative value. The underlying values need not be adjacent, e.g., *decreased* and *increased* can be joint to a new value *pathologic*. These combinations are defined for each abstract data-type which ensures flexibility but avoids redundancy.

Temporal Interpolation. The truth-persistence function is binary. It is modelled by the definition of two durations per parameter, a *trust period* following the measurement and a *retrospective trust period* preceding the measurement during which the parameter is considered to hold the value measured. If the trust period of a measurement overlaps the retrospective trust period of the next measurement, they are reduced proportionally until they meet.

A complex form of defining a global truth-persistence function is the application of the *Spread* algorithm described in section 4.4.4.

Temporal Pattern Matching. The most common form of queries is the *predicate query* implemented by the *parameter proposition*. Due to its usage in conditions triggering plan state transitions, it has different semantics than its RÉSUMÉ pendant: If there is no interval in the past matching the proposition, the system waits until such an interval occurs. Only if it is impossible that a matching interval will occur (e.g., since the latest finish time already passed) *false* is returned and the waiting is terminated.

The result of *set queries* are used as means for further queries such as asking for the start of the first of these intervals. Sets of intervals cannot be stored and handled explicitly. *Value queries* in Asbru only return the current value of a parameter at the time of the query, there is no way to obtain the value of a parameter at some past time[5], but its existence can be queried using predicate queries.

The Tzolkin architecture resembles the original design of the Asbru execution engine [93] described at the start of Section 4.1. However, the solutions described in this thesis transformed the separate mediator between separate module blocks by a shared management unit which continuously triggers those bits of the system which need to take small steps to incrementally enrich the history of abstractions. This approach is necessary to meet the requirements of high-frequency domains, while in low-frequency domains, database-oriented approaches like Tzolkin, Chronus, and RÉSUMÉ balance the longer response time to queries with the advantage of distributing the abstraction process over arbitrary spans of time.

To handle complex constraints possible in CAPSUL, Asbru needs to be combined with sliding time windows and utility functions to handle calendar references.

[5]The delay module described in Section 4.2.2.5 provides a work-around.

2.3 Neighbouring fields

This section briefly introduces a range of fields and gives a few examples of work in them. Some of these fields contribute indirectly to the work described in this thesis. Some of them present alternatives for it. Of the latter, some can be integrated with my work, as described below.

2.3.1 Statistics

Statistics play several roles in temporal data abstraction. See [3] for an introduction to the field.

Descriptive statistics provides abstractions from a group of values, such as maximum, minimum, mean, median, and centiles. Computed at regular intervals, they can abstract high volumes of data into a stream of lower volume and more significance for the next processing steps, if the definition of the abstraction matches the view of domain experts on the data. Applied to the content of sliding time windows, they are included in the range of abstractions described in Section 4.3.2.

Statistical modelling plays an important role at design time of temporal data abstraction systems. On the one hand, the knowledge engineer selects one or more suitable algorithms and sets their parameters to reflect the information received from domain experts. On the other hand, domain experts can be asked to annotate sample data sets with the desired output.

Statistical methods, like multiple regression, can then create a sound model of the dependencies between the inputs of the abstraction process and the output or target function (or dependent variable in statistics terminology), which is the markup by the domain expert. Comparing them to the manually crafted abstraction algorithm will be instructive even though it need not be identical. Domain experts may have good reasons to suggest rules which seem to contradict the sample data, but such contradictions need to be discussed carefully. Other statistical methods allow the exploration of the quality of several candidate models to guide the development of data abstraction schemes.

2.3.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) [70] are popular means for mapping multiple quantitative inputs to a single qualitative output. They are computational expensive and mapping their internal configuration to domain knowledge can be difficult. However, some studies report better results than for logistic regression, e.g., in [50], while others only show noteworthy differences in the ROC[6] curve but no general advantage of either approach over the other [151].

ANNs generally require a carefully chosen set of training data to configure them. In contrast, clinical guidelines specify the details of data abstraction more or less ex-

[6]Receiver Operating Characteristic. The ROC curve plots the true positive rate (sensitivity) over the false positive rate (1-specificity) [185]

plicitly, which is difficult to translate into ANNs (and the autonomous learning is seen as their main advantage).

Still, the following example is instructive in the context of this thesis, since it combines ANNs with non-ANN artefact detection, and it deals with high-frequency data similar to the ideas presented in this thesis.

Zoubek [183, 184] build an automatic system for the classification of polysomnographic recordings into different sleep/wake stages. Recordings comprised Electroencephalography (EEG), Electrooculography (ECG), and Electromyography (EMG). These signals are prone with artefacts. The frequency of all three signals is 128 Hz.

In a first step, artefacts are detected using the PRANA® package and eight different rules for eight different classes of artefacts, such as loss of signal or 50 Hz power line artefacts. The recording is broken into 20 second segments. If more than two artefacts are detected in a segment, the whole segment is discarded. The EEG is considered the most important signal. If it is discarded, this segment is not analysed at all. If one or both of the other signals are missing, the classification is still performed with the non-disturbed signal. This leads to four different settings (EEG alone, EEG + EOG, EEG + EMG, all three signals). About 25% of the segments showed at least one artefact, but only 20% of these segments showed an artefact in the EEG and had to be discarded, while the switching to alternative input combinations salvaged 80% of these artefact-affected segments.

Next, a band pass filter was applied to each of the three signals. The measurements in a segment ($128 \times 20 \times 3 = 2560$) were considered too numerous as input for automatic classifiers (ANNs). Therefore, 33 features were extracted from each segment.

The first group is constituted of features often referred to in the literature, namely the relative power of the EEG signal in five different frequency bands, the relative power of the EMG signal in a high frequency band, and the highest frequency below which 95% of the total spectral power is located, for all three inputs. The second group is a set of standard statistics metrics, computed for each of the three signals: standard deviation, skewness, kourtosis, upper quartile, and entropy. The The third group consists of three parameters suggested by Hjorth [72], namely activity, mobility and complexity. Activity is equal to variance, mobility is a measure for the mean frequency. Complexity is the ratio of mobility and the first derivative of the signal.

Then, a combination of Sequential Forward Selection, Sequential Backward Selection, and ANNs (multi-layer perceptrons) was used to find the optimal number of features. This was performed for each of the four combinations of available signals, i.e., if all three signals are available, then features are taken from all three, but if one is missing, features are taken from the other two. Therefore, the classification algorithm needs not deal with missing values. The total number of features is always seven. Based on them, an Artificial Neural Network assigns one of five sleep stages to each segment.

The overall classification accuracy is 85.6%. A single classifier without prior artefact removal achieved 83.2% on the same data. For those 20% of the input data where at least one artefact was removed and only two or one or the three input signals were available, accuracy was the described system was still 81.2%.

The above example showed how combining a rich set of abstractions in more than one step to merge traditional medical knowledge with insight gained by data analysis can lead to results not obtainable otherwise. The work in this thesis follows a similar approach, albeit using statistical abstraction instead of neuronal networks.

2.3.3 Planning

The AI field of planning deals with the optimal arrangement of actions (called plans) which have pre-conditions (which must be fulfilled before the action can be taken) and post-conditions (which are assumed to be true after the action has been taken). Unfortunately, in the medical domain, the effect of an action, i.e., the post-condition, can rarely ever be given with certainty, which greatly complicates the reasoning process connected to the quest for an optimal (or possible) solution.

Anselma and Montani [9] present an overview of planning in the context of guideline execution.

As for other guideline representation languages, the ideas of Hierarchical Task Networks (HTN) [52] are present in Asbru, although dormant in part.

Asbru does represent a guideline as a hierarchy of skeletal plans, implementing the stepwise refinement generally characterising HTNs, with conditions on each level of the hierarchy.

The semantics of the setup-precondition specify that the execution engine is to match it against effects and intentions of other plans. Matching plans are to be activated which would result in fulfilling the setup-precondition, if and when the defined effects take place or the intentions are achieved in the real world.

However, in past project, acquiring solid information on intentions and effects proved unexpectedly difficult, because they are rarely stated explicitly and precisely in guidelines or protocols. Without a complete set of plans for which effects or intentions are known, attempts to matching setup-preconditions against them cannot succeed.

Our collaboration also showed that in the medical field, high degrees of control over all details of plan execution are generally desired. We therefore focus on synchronisation and adaptation of plans.

The first is implemented by monitoring the parameter propositions in the conditions of plans. The second is modelled as switching between plans as mandated by the patient condition.

Asbru plan hierarchies deal with a single patient by definition. Approaches which deal with patient groups may find useful applications of optimising the resource utilisation by arranging treatment schedules on a multi-patient level. This is, however, far beyond the scope of this thesis.

Chapter 3

Problem Description and Objectives

This chapter develops a problem description from the domain description in Section 1.1 and the state of the art described in Section 2. It also states concise objectives to be met by the solutions described in detail in Section 4. Section 5.3 discusses how these solutions meet the objectives.

3.1 Objectives related to data abstraction

This section develops objectives regarding temporal data abstraction, while the next subsection focuses on guideline execution.

3.1.1 Coping with noisy and missing data

Health care today utilizes a large variety of technical equipment to exactly monitor the patient state. However, these measurements suffer from distortions caused by broad range of factors. Some examples are:

- Movements of the patient can lead to movements of a sensor which again disturbs the measurement during the time of movement.
- External events such as care activities influence the patient's condition. Without knowledge about these activities, the resulting changes in physical parameters of the patient may appear as unexplained artefacts in the data.
- Measurements of the same physical entities by different means often do not provide the same value due to inherent bias or lack of precision of each technology. In particular, laboratory tests provide more precise measurements at a low frequency and higher cost (in terms of labour and patient discomfort) while monitoring devices provide continuous information with little patient discomfort but also reduced reliability.
- Measurement may be missing due to sensor failure. Also, the responsible person may simply forget to take a measurement, e.g., in the field of self-monitoring in diabetes.

The reaction to missing or erroneous data can be manifold, depending on the kind of information to be abstracted from the data. Bad measurements can simply be erased, they can be replaced by estimates, or the abstraction process can be designed in a way which is robust with respect to disturbed input. Clearly, the third choice is the most advantageous and difficult.

A very important issue in medical data abstraction – and badly affected by noise – is the abstraction of qualitative values from quantitative ones, e.g., to decide whether saturation of oxygen was sufficient in the previous minute or not, based on measurements delivered every second. Here ignoring a few measurements is no problem, but inserting gaps in the output is not appreciated since we later want to ask questions like "For how long was oxygen low?", which will be rendered impossible by inserting small gaps frequently, just because of single bad values in the raw data. On the other hand, the amount of uncertainty must be considered, if it is increasing during some phases.

The resulting objectives can therefore be divided into two groups: Integration of existing approaches into the framework for data abstraction, and creation of a new solution to the problem of abstracting qualitative values from noisy quantitative raw data.

Existing methods for to detect and remove faulty measurements range from static measurements such as fixed boundaries (minimum and maximum) for the value itself and for the difference to the previous value to more advanced boundaries that can be derived from a set of previous measurements by statistical analysis. Another approach is to define dependencies between different parameters, in terms of rules [75].

Objective 1. To combine existing approaches such as comparing different input channels, averaging and comparing against the average, into the framework for data abstraction, error detection and repair.

The sample application for the abstraction of steady qualitative values is the artificial ventilation of neonates. Here, it is important to react to good quality input swiftly when it is available, while exhibiting a robust or defensive behaviour during phases of significant noise, and to change gradually between them.

Objective 2. To find a new solution for abstracting steady qualitative values from quantitative data containing varying parts of noise, such that the threshold used to suppress undesired changes in the output depends on the current amount of noise.

3.1.2 Flexible definition of time windows for statistical analysis

Descriptive statistics are a powerful and widely used and accepted means to obtain high level descriptions from a large set of measurements. However, analyzing a whole recording, e.g., one night as a whole, does not allow the comparison of parts with each other to detect subtle trends or patterns.

Separating the measurement into equal slices of, e.g., 1 hour does allow this, but there are some drawbacks of this approach.

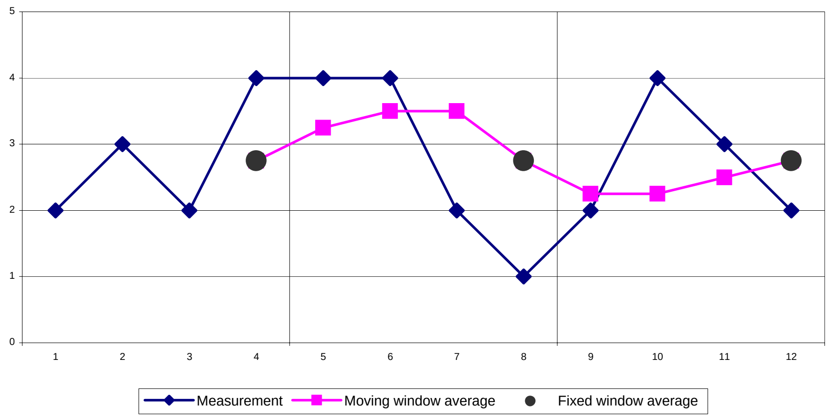


Figure 3.1: Comparing the average of 4 measurements for fixed (disks) and sliding (squares) time windows. The average of each of the 3 groups of four measurements is 2.75. However, the average measured of a time window of 4 measurements which is advanced by 1 clearly reflects the three high values at time 4 to 6 and the three low ones at time 7 to 9. If the task were to detect time periods for which the average was greater than 3, then looking at the average of fixed groups of measurements would not find a matching episode, while using a sliding time window does.

- An occurrence of the pattern searched for can be split across two slices and each of the resulting halves may be fail to be detected since it does not appear significant enough within its slice.
- The analysis of a time slice is performed at fixed times with considerable intervals in between. A pattern at the start of a time window is only detected at the end, which may cause undesired delays, e.g., in alarming.
- Reducing the time window size reduces this delay, but increases the risk of split patterns.

Figure 3.1 illustrates this. In the example, the average for a time window of length 4 is examined. If it is greater than 3, an alarm is to be triggered. Grouping the measurements in fixed time windows of size 4 fails to detect the increase on the border between the first and the second time window. In contrast, the average calculated from a sliding time window with step width 1 exceeds this threshold at 3 positions.

A standard solution to this problem is the application of a moving time window which slides steps smaller than its width along the time axis as shown in Figure 3.1. This approach must be integrated in the framework of data abstraction and various statistical evaluation methods for the contents of a time window must be provided.

Objective 3. To integrate the aggregation of input in sliding time windows of freely defined size and step width into the common framework, together with evaluation functions such as average, mean, quantiles, and linear regression.

3.1.3 Utility functions

To solve real world problems, each framework for data abstraction needs simple but indispensable abstractions such as arithmetic and logical combinations. Consider, e.g., the above example of analyzing different sliding time windows. We can abstract the average and trend of different time windows. But to define the abstraction "average of previous minute lies two or more units above the average of the previous hour and the trend is negative" we need addition, comparison, and the Boolean and-operator.

Therefore, the following groups of operations must be added to the (much more) sophisticated abstraction described in other sections:

- arithmetic operations,
- logical operators, and
- comparisons.

Objective 4. To integrate utility functions implementing arithmetic and logical operators as well as comparison operators with the other modules.

3.1.4 Integration of abstraction modules into a uniform framework

It is clear, that all abstraction modules described in this thesis must interoperate. To this end, they must be part of a framework in which they are as interchangeable as possible, given their different nature.

This framework must ensure that the input is processed with minimal computational effort, i.e., the logic of abstraction modules must only be consulted if new input is available for this module.

Objective 5. To integrate all the algorithms presented in this thesis in a framework permitting the greatest possible freedom in combining them. Data flow in this framework must be organised in such a way as to minimise computational effort.

3.2 Objectives related to guideline execution

As argued in Section 2.1.3, Asbru is the formalism of choice to model temporal patterns in connection with guideline execution. It also excels in temporal aspects of plans. Therefore, the following objectives are aimed at implementing the execution of Asbru plans within the framework described above.

3.2.1 Online-algorithms for the detection of temporal patterns

Asbru contains powerful means to describe the temporal extent of parameter-value pairs, plan activations, etc., but previous implementations were not suitable for high-frequency domains.

For such a domain, an implementation must process incoming data measurement by measurement and output any abstractions immediately. It must meet both the demand from plan execution and data abstraction.

Plan execution uses temporal patterns in the conditions of plans. Data abstraction uses temporal patterns as most complex form of temporal abstraction. The most important parts of the Asbru element *temporal-pattern*, which needs to be implemented by the proposed solution, are the following.

parameter-proposition describes a value for a parameter and an interval during which this value must hold, as well as a context which must be given. It is the core unit of environment monitoring.

plan-state-constraint describes a plan state for a plan and an interval during which the plan must be (or have been) in that state. It resembles the parameter proposition with the main difference that plan states, which are changed internally by the plan execution unit, are monitored instead of parameters reflecting the external world, and the minor difference that no context is given.

temporal-constraint defines the qualitative relation of intervals which again are defined by temporal patterns.

constraint-combination defines Boolean combinations such as conjunction and disjunction for temporal patterns.

count-constraint contains a temporal pattern and a minimum number of occurrences for this temporal pattern. Only if this number is reached, the count constraint is fulfilled.

simple-condition defines a relation between two instantaneous values. These values can be parameters or variables. In the first case, the current value of the parameter is accessed, so there is no abstraction in addition to supplying the abstractions defined for this parameter.

The implementation of the semantics of these Asbru elements must be compatible with the data flow in the abstraction framework, and with the implementation of plan execution.

> **Objective 6.** To find implementations for the Asbru elements *parameter-proposition*, *plan-state-constraint*, *temporal-constraint*, *constraint-combination*, *count-constraint*, and *simple-condition*. They must be integrated into the framework of temporal data abstraction and guideline execution, and handle high volumes of data efficiently.

3.2.2 Integration of plan execution

Plan execution in the context of this thesis refers to the implementation of the semantics of the Asbru plan. The most important parts, or knowledge roles, concerning execution are conditions and plan body. The functionality of conditions is covered by the objective above. The plan body organises subplans (or child plans) in one of the following:

- Sequential: all subplans are performed as listed.

- Parallel: all subplans are started at the same time.
- Unordered: subplans are started whenever they are ready.
- Any-order: only one plan is active at any time, the order is not predefined.
- Cyclical: the subplan is performed several times.

Another important aspect of the parent-child relation is the propagation of success and failure, given in the *continuation-specification* and *propagation-specification*.

The implementation of this part of the Asbru syntax must follow the same principles as the data abstraction – execution must be input-driven, avoiding active polling (which is used in other implementations of Asbru) because it is not suitable for high-frequency domains.

Objective 7. To translate the semantics of Asbru plans to the process-logic the abstraction framework.

3.2.3 Bridge from Asbru to abstraction modules

The ultimate aim of the execution unit is to execute plans specified in Asbru. Therefore, in addition to the modules described above, a mapping of all relevant Asbru elements to modules which implement their semantics must be described.

In terms of implementation, this means that an Asbru compiler must read the Asbru plan library, create instances of suitable modules, and link them such that the flow of data from the input devices through data abstraction modules to monitoring modules and from there to modules representing Asbru plans results in a system behaviour which is compliant with the semantics of Asbru.

For example, a user-performed plan with a *filter-precondition* stating that parameter A must be higher than 5 for 1 hour will create something like the following.

- A module representing input A;
- A module representing the constant 5;
- A comparison module outputting *true* if the input coming from the module representing A is greater than the input comming from the constant module (i.e., 5);
- A monitoring module which receives the output of the comparison module as its input and which outputs *condition fulfilled* when its input is *true* for one hour without interruption.
- A module representing the user-performed; Its input named *filter-precondition* will be connected to the output of the monitoring module.

Objective 8. To map all Asbru elements relevant for plan execution to one or more modules implementing the semantics of this element.

Chapter 4

Solutions

This chapter takes a bottom-up approach to describing the solutions to the problems described in the previous chapter. They are ordered as follows.

Uniform framework below describes the foundations and design of a framework in which the work described in the other sections is integrated.

Utility functions on page 66 describes a set of simple but indispensable abstractions. In combination with more complex ones described in other sections they are needed to meet the demand from practical applications.

Multiple sliding time windows for statistical analysis on page 73 describes abstractions based on subsets of the data defined by a time window which progresses along the time axis.

Coping with noisy data on page 79 shows abstractions specifically designed to arrive at the most reliable abstractions possible in the presence of noise. Many of these solutions are combinations of abstraction modules introduced in Sections 4.2 and 4.3.

Online-algorithms for monitoring temporal patterns on page 90 contains several algorithms to implement various complex abstractions which are fundamental to the plan representation language Asbru.

Integration of plan execution on page 134 describes abstraction modules which in fact implement Asbru plans. The achievement of mapping all Asbru plan functionality to state machines which take inputs and produce output in the form of abstract data points is key to the seamless integration of temporal data abstraction and plan execution.

Bridge to Asbru on page 148 describes in detail how Asbru elements map to the abstraction modules described in the section before.

4.1 Uniform framework

This section describes the framework for the solutions of subproblems presented in the following sections. The foundations for the design of this framework are the original design of the Asgaard runtime system for executing Asbru plans, and the nature of the data and the abstraction process.

The historic background

One of the main characteristics of the Asgaard runtime system is *continual environment monitoring* [93]. This means that throughout all phases of plan execution, the current state of the environment is observed and changes in the environment influence the execution of plans. This means that the patient state is monitored and changes in the patient's condition change the way the guideline is executed. Originally, we distinguished *plan synchronization*: the start and end time of a plan or treatment step are synchronized with the patient's condition, *plan adaptation*: the modalities of a treatment steps depend on measurements taken from the patient, and *replanning* where the original plan is discontinued and new plans (e.g., different treatment options) are selected.

This lead to the following principal design.

- New measurements arrive at the data abstraction unit whenever they are available (independent of plan execution). If appropriate, they are abstracted to higher-level concepts. These abstractions are specified in the domain definition part of the Asbru plan library.
- The monitoring unit receives the output from the data abstraction unit and continuously searches for temporal patterns such as high fever for more than 1 hour in the streams of data. It is informed by the plan execution unit which patterns are relevant. If a pattern is detected, the monitoring unit informs the plan execution unit.
- The plan execution unit controls the states of the plans described in the Asbru plan library. Changes in these states require conditions to be fulfilled. Conditions contain temporal patterns. Whenever a plan state transitions becomes possible (because the plan in question reached the relevant state), the plan execution unit sends a request to detect the temporal pattern in this condition to the monitoring unit. As soon as the pattern is detected, the plan execution unit is informed. It then changes the state of the plan whose condition is fulfilled.

The new idea

The potential large history of input data prohibits repeated searches for patterns from scratch. Instead, on-line algorithms must be found to detect temporal patters as soon as they occur, based on continually arriving input from the abstraction unit. This means that the internal structure of the monitoring unit must be similar to that of the abstraction unit.

If an implementation of plan execution can be found, which is compatible with the scheme of data abstraction and monitoring, then the matching process of requests from the execution unit and found patterns in the monitoring unit is no more needed. The monitoring unit feeds its output unprompted to the execution unit. Monitoring and execution unit then share their architecture with the abstraction unit, which means that all steps are implemented within a uniform, seamless framework.

The abstraction unit is implemented by modules described in Sections 4.2 to 4.4. The monitoring unit is implemented by modules described in Section 4.5. The plan execution unit is implemented by modules described in Sections 4.6 and 4.2. However, there is no strict boundary between these parts. In contrast to the original design, output from the monitoring and plan execution process can be fed back into the abstraction process, multiplying the power of the resulting system.

4.1.1 The nature of data

Before designing the framework itself, it is necessary to explore the nature and features of the data to be handled within this framework.

4.1.1.1 Sources of data

There are three sources of data: user input, measuring devices, and storage devices. In theory, storage devices are not a source of data, but in practice they often serves as such - the data is collected by other software and served in to the data abstraction process in the form of files or databases.

Each source of data brings in different types of error. User input is sometimes distorted due to human errors, it is often missing due to time constraints or motivation of the humans performing the measurement (e.g. in emergency care or Diabetes), and it can be substantially delayed. Automatic measurements suffer from various technical problems such as loss of calibration and disturbance of measurements due to patient movements or interventions by care personal.

Input from file storage such as patient data records often suffers from incomplete data and the general inability to obtain missing information. In an interactive environment, where the patient record is accessed during patient encounter, data can interactively be complemented as required. However, in batch processing historic data to evaluate a guideline model, or to verify a hypothesis, such supplements are not an option. This can render significant portions of stored data useless, unless the protocol model is robust enough to handle these cases.

4.1.1.2 High-frequency versus low-frequency

In high-frequency domains, data arrives, e.g., once per second on a typically regular basis from a monitoring device or file. In the second case, data is typically entered by the user or obtained by potentially complex queries to data bases at a rate of several times per day or week.

Due to the fact that measurement are automatic, there is a tendency to have many measurements available in high-frequency domains, e.g., to judge the oxygen supply of the previous minute from 60 measurements. Still these need not be too many since the quality of the input signal can be bad enough to require a large sample of raw data to make any safe conclusions.

Low-frequency domains often suffer from missing information or to sparse measurements. At the same time, the measurements are not always reliable either (e.g., for self-monitoring in Diabetes). In many cases, single measurements in this domain constitute valid information by themselves although there are counter examples. E.g., a careful measurement of body temperature reliably gives the amount of fever for about half an hour before and after the measurement. In contrast, a single measurement of oxygen saturation in blood using pulsoximetry does not even give a reliable information, even for the second of measurement, only several measurements together prove whether there was an artefact or not.

Abstracting this situation, we could say that high-frequency domains suffer from a lack of information and at the same time from a flood of information while low-frequency domains suffer from a lack of data. Therefore, in high-frequency domains major effort goes into data validation and abstraction of reliable information from unreliable measurements while in low-frequency domains major effort goes into the care flow process to ensure that the right measurements are taken at the right time and that they are entered into the system with minimum loss or distortion of information.

Loosely coupled to this distinction is the separation by mode of patient encounter. A patient may be available all the time during treatment, e.g., in a hospital. In another setting, the patient may be absent most of the time an only encounter the physician for routine control or in case of any emergencies. This is the case for chronic diseases which do not require hospital stay.

Closely coupled to the distinction of patient availability is the degree of responsibility. While in an intensive care unit the medical staff handles all steps to obtain any needed information, outpatients with chronic diseases perform the monitoring task themselves. Therefore, in the first case there is a host of (potentially erroneous) data and the main issue is find those pieces of information in the which are important (e.g., triggering an alarm if a disadvantageous trend is observed) while in the second case motivation of the patient is an important precondition of successful patient monitoring.

It follows from the above, that the distinction between low-frequency and high-frequency of data is not the most important distinctions between these domains, but it is the common term to distinguish them and so I keep these labels in this document. By the same token we could talk about automatic versus manual data-domains or permanent versus sporadic patient availability domains.

Table 4.1 summarizes the characteristics of the two types of data. Note that the gap in data frequency is not definite, it is only there since there is little data at such a rate.

4.1.1.3 Quantitative versus qualitative

In any case, data can be both quantitative, i.e. numeric, or qualitative, i.e. symbolic. An example for the first is fever in degrees Celsius, while fever expressed as “high”, “moderate”, or “no fever” is an example for the second case.

Asbru defines a series of data types for numeric data, e.g., length, volume, but also *time* (i.e., temporal distance) and *date* (i.e., absolute point in time). The data abstraction unit only distinguishes between integer and real values, but to the interface to the plan library converts these simple representation into more complex Asbru value objects.

	High frequency	Low frequency
Interval of measurements	1 ms – 1 s	hours to months
Mode of input	automatic	manual or automatic
Patient availability	permanent	occasional (permanent for self monitoring)
Major issue	transforming unreliable snap shots into useful information	ensuring sufficient availability of data

Table 4.1: Typical characteristics of high-frequency and low-frequency data.

Purpose	Automatic plan execution, patient monitoring by humans, knowledge discovery
Mode of operation	real-time or playback
Nature of values	Quantitative or qualitative

Table 4.2: Characteristics common to both high-frequency and low-frequency data.

4.1.1.4 Continuous versus discontinuous

Data can also be separated into continuous and discontinuous data. However, seen from the implementation side, the differences are not significant. One definition is that continuous data is entered on a regular basis. This is true for data delivered from a monitoring device as well as for laboratory tests in chronic diseases. Both share a lack in precision of the time point of measurements – neither are 1 Hz measurements delivered 1000 ms after the previous one, nor are routine tests performed exactly on the day they should. Both domains share the lack of data – either from sensor failure or from patient unwillingness or patient management faults. There is data which is strictly discontinuous – single measurements that cannot be compared to each other – but they can be treated by data abstraction just as continuous values that are surrounded by huge gaps if these algorithms are subtle enough. If they would not, missing data would lead to wrong abstractions.

To summarize, Table 4.2 shows the dimensions of data abstraction that need to be considered in parallel in the rest of this document.

4.1.1.5 Valid time and transaction time

Each measurement is taken at a certain point in time. For some applications this time may not be important but in most cases it is. Queries for data which do not consider the temporal aspect can easily be mapped to such with a temporal dimension which is arbitrary, so they are not considered in the rest of this section. Similarly, measurements that do not come with a precise time of measurement are fitted with the time at which they first appear to the system (i.e., the data abstraction unit.

We distinguish two times for each measurement: The *valid time* and the *transaction time* [132]. The valid time is the time at which the measurement was originally taken and at which the measured value is known. The transaction time is the time at which the value was entered into the system.

Input from users is often considerably delayed, i.e., there is a considerable difference between valid time and the transaction time. This causes particular problems in domains with high-frequency data, which must be synchronized with the delayed low-frequency data. An example is the measurement of blood gases with a fast but inaccurate pulsoximeter and by slow but precise laboratory test in parallel to validate the first with the second.

Input from monitoring devices can be delayed too, although on a smaller scale. In a practical application of a pulsoximeter which is expected to deliver one data record per second, we found that for 10 % of all seconds during measurement there was either no or two records. This is explained by the varying work load of the device's processor. It is not considered harmful because the latency of a human in processing the delivered information exceeds the maximum latency of the device which is about one second. Still, on the implementation level, this property of the raw data can cause additional effort.

In any case the maximum difference of valid time and transaction time must be defined for each parameter to limit the uncertainty in the abstraction process. Without waiting for this delay, irrevocable measures could be taken based on wrong information. In the following, this difference is called the *maximum data delay*.

4.1.1.6 Data validity over time

Another aspect is the time between two measurements. It is clear, that a measurement is not invalid the moment after it is taken, but it is also clear that it cannot be valid without a temporal limit. So the decay in trust into the measurement needs to be modelled. This can be done by introducing an additional dimension – the trustworthiness of each measurements, but this would dramatically increase the complexity of the abstraction process, especially for values which are derived from several sources. In practice, an exact modelling of the trust into a value over time is neither necessary nor possible. For our purpose, we opted for a single parameter, the *trust period*. It defines an interval starting at the valid time during which the measurement is considered valid. After it, the measurement is considered invalid, i.e., until there is a new measurement available, the value of the parameter is considered unknown.

Besides this trust period, there is also a *retrospective trust period*. If a certain parameter has a certain value at a certain time, it is very likely that it has the same value shortly before. The interpretation of shortly strongly depends on the parameter and the domain, but it will always be a considerable amount of time relative to the usual frequency of measurements. E.g., for a parameter measured every second it might be 1/4 of a second while for fever measure 3 times a day it may be one hour.

The relation of trust period and retrospective trust period varies depending on the mode of measurement. If measurements are performed automatically or otherwise without influence of the patient on the timing, then both periods will be equal. Examples are oxygen levels in blood measured each second and blood analysis performed at health checks which are scheduled on a regular basis, e.g. once per year. If the patient takes a new measurement after experiencing a change, as it is the case for fever, then the reverse trust period will not be as long as the trust period. If the parameter is a symptom the patient can observe easily like coughing or itching, then the reverse trust period will shrink to nearly zero, since there cannot be an unobserved change (except for biases in the patient's cognition).

For pragmatic reasons, the trust period will often be extended beyond what would be justified by an exact model because one often prefers to use an old but outdated information to total blindness due to lack of any information.

A third aspect is brought in by the fact, that measurement are often performed rather regular, which gives some guarantee that after one measurement there is a time span during which no further measurement is expected. This means, that for this interval, the value can be considered the finally valid one. This interval is called the *minimal interval between measurements*.

4.1.2 Implementation decisions

This section briefly lists the choices taken for the implementation of various issues described above.

4.1.2.1 Time of abstracting

Obviously, there are two options for calculating abstractions from the input: immediately or on demand. Demand is defined by the plan execution unit or by the user depending on the application.

For immediate abstraction, the effort is proportional the number of different abstractions, i.e., the number of parameters abstracted from the raw data. This number can be considerable in plan execution, since it not only comprises those abstractions explicitly defined in the plan library, but also any reference to the data in any condition in the plan library.

On the other hand, calculating only those abstractions which are needed at a certain moment brings along the undesired possibility that abstractions are calculated multiple times during plan execution if they are needed several times with breaks in between. As a remedy, all abstractions calculated on any occasion should be stored, but if their calculation is discontinued when they are not further interesting, then managing the information, which pieces of abstraction have already been calculated becomes difficult.

In the case of patient monitoring, additional requests will be rare, because all abstractions are defined in the protocol or guideline on which the treatment is based.

While storing all abstractions is advisable from the point of saving computation time, it becomes a problem when looking at storage space, at least in high- frequency domains. A single parameter measured once per second needs more then 1 MB in 24 hours (see the implementation section about the exact details). Therefore, storing all abstractions may not be feasible in such a domain. A compromise can build on finding the "important" nodes in the stream or graph of abstractions and storing only them, but the definition of "important" itself is difficult.

The mentioned storage shortage can only occur for high-frequency streams of single point data. Temporal abstractions always summarize several, if not many of them into one (complex) data item. Therefore abstracting the raw data and working on the

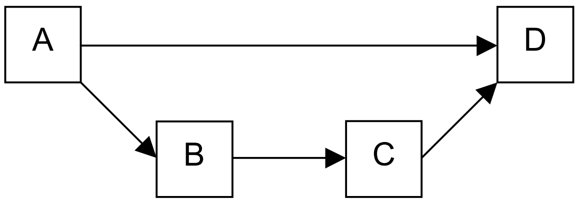


Figure 4.1: Sample dependency graph.

abstractions reduces the storage problem. For low-frequency domains these problems rarely occur since the data collected during the life of a diabetes patients is smaller than that collected for a patient in an intensive care unit per day.

4.1.3 Overview of principal parts

In the abstraction process, various classes or groups of classes take over different roles. The following describes these roles and thereby gives a first introduction to the class hierarchy.

4.1.3.1 Management of timing

The *data flow* through the abstraction modules forms a directed graph. It is frequently the case that two branches of different length meet at some node as depicted in Figure 4.1. In such a case it is important to observe one of the few correct sequences of processing the abstraction steps, e.g., A-B-C-D, and not to process D before B or C. Note that neither depth-first nor breadth-first search would produce the right sequence but more sophisticated graph analysis is needed.

Therefore, the abstraction modules cannot directly feed the data to each other. Instead, a managing module must be implemented, which analyses the data flow before processing starts and which for each input value activates the abstraction steps in the correct order.

A second complication arises from the fact that different signals may have *different frequencies* and also abstraction modules can have a frequency of output different from their input. Even worse, no source of data nor abstraction module is obliged to operate at a fixed frequency at all. This again calls for careful control by the management module.

The same problem occurs when comparing manually entered measurements with those automatically entered. The first ones will come many minutes later then the latter. The result can only be that the signal abstracted from both inputs will be as delayed as the manual input.

4.1.3.2 Source interfaces

Independent of the source of data it needs to be stored in an intermediate format before it is processed. Qualitative values are mapped to integers, while quantitative values are stored in real numbers (`float`). In both cases, a time stamp is attached to each measurement.

4.1.3.3 Abstraction modules

The heart of the abstraction unit are, of course, the abstraction modules. Each takes input in the form of single data points or complex objects and process it in some individually defined way. Each abstraction step, i.e., the processing of each data point is triggered by the management module. At the end of processing a data point, the module returns the result of abstraction to the management module which passes it on to those other modules which are connect with the module in question. The abstraction modules need not produce output for each step. An example for such an abstraction is to sum up the input and calculate the average in certain intervals, e.g., after 10 input data-points.

Each abstraction module implements the methods `newData` and `timeout`. Both are called by the management module. The first is called when a new data point is ready for processing. Modules with multiple input sources (e.g., add) are only called once for each value of valid time. If some of the sources did not supply new values, the management module supplies the most recent values from previous processing steps.

Some abstraction modules react not only on new data but also on the progress of time. In these cases, they call the either `setPreAlarm` or `setPostAlarm` of the management module with the valid time at which the module needs to be activated independently of the data flow. For each step in valid time the management module first calls `timeout` for all *PreAlarms* for that time stamp, then it calls `newData` in a suitable sequence and finally it calls `timeout` for all *PostAlarms* for that time stamp. The differentiation of the two alarm varieties is necessary for the implementation of convex and concave time intervals.

Both `timeout` end `newData` return a new data point output by the module in reaction to the event or input, or `null` if no new output was produced.

4.1.3.4 Data types

While input data is restricted to qualitative and quantitative values, data points resulting from abstraction need not be single measurements, they can also contain complex structures. But they are all subclasses of a common root class (`AbstractDataPoint`) and all contain the field `validTime`. This way the management module can handle them in a uniform way, while different objects can receive different forms of input and produce complex output within this framework. The following data points are currently implemented.

FloatDataPoint contains only a `float` in addition to the valid time. It is the standard class to store quantitative measurements. `Float.NaN` is used to declare the value unknown. ∞ and $-\infty$ are represented by `Float.POSITIVE_INFINITY` and `Float.NEGATIVE_INFINITY`.

IntDataPoint contains a int instead of the float. It is used for qualitative values including Booleans. Qualitative values are numbered starting with 0. For Booleans, 0 is *false* and 1 is *true*. -1 marks an undefined value in both cases.

TimeDataPoint contains a time value, i.e., a data point in which the value describes either a data or a duration.

FloatDataPointSeries contains an array of arbitrary data points representing a sequence of measurements taken from the same input. It is produced by modules such as time window.

RegressionLineDataPoint describes a linear regression model computed for the content of a time window at a specific point in time. The RegressionLineModule desribed in Section 4.3.3.2 translates each FloatDataPointSeries to a RegressionLineDataPoint.

EpisodeDataPoint contains information about an episode, which is an interval matching a time annotation, as described in Section 4.5.2.

PlanModuleOutputDataPoint contains the plan state, signals to the child plans of the sender, and optional numeric output from the plan module sending it. It is mostly used for communication between plans. See Section 4.6 for details.

4.2 Utility functions

The more sophisticated abstraction functions are glued together by a set of simple utility functions. While their description might seem trivial, it is an important feature of the described framework to contain such simple complements to the sophisticated modules, because only the combination of both can satisfy the demands met in practice.

Since their description also provides a gentle introduction into the complex consideration of timing and side considerations which occur in temporal data abstraction, I put their description before, and not after, the description of the sophisticated modules.

Where appropriate, details of the implementation are given. Each abstraction module is described in terms of its input, output, and timing. Timing is only mentioned where output is not produced in reaction to each input data-point becoming available.

Where appropriate, arguments of the algorithm are described. The difference between inputs and arguments is that inputs always change dynamically and are subjected to preprocessing while arguments specify the form of processing and are constant during the time of processing. This does not mean that they must be constants in the Asbru plan library. They can be variables which are evaluated before monitoring starts. The values of arguments are often given in the plan library, while the value of inputs is always delivered by other modules (or data sources).

4.2.1 Arithmetic operations

Based on the number of inputs per module, arithmetic operations can be grouped into n-ary calculations (with any number of inputs, typically greater one), binary inputs (with exactly two inputs), and unary calculations (based on only one input).

4.2.1.1 N-ary calculations

This section summarizes the behavior of add, multiply, minimum, and maximum.

Input. These four calculations take two or more values from different channels at the same time (FloatDataPointSet). Note that minimum and maximum can be applied on a *sorted time window* of a single channel, too – see Section 4.3.2.1. For minimum and maximum, the input can be qualitative or quantitative, for add and multiply it can only be quantitative.

Output. The sum, product, minimum, or maximum of the most recent values of the input channels. If at least one input value is undefined, then the output is undefined, too.

Timing. Whenever new input is available from at least one channel, new output is generated. If the trust period of an input channel is over without a new value available, recomputation of the output is triggered using the undefined value for the missing input. This leads to undefining the output. Compare figure 4.2.

4.2.1.2 Binary calculations

This section summarizes the behaviour of subtract, divide, root, exponent, and logarithm.

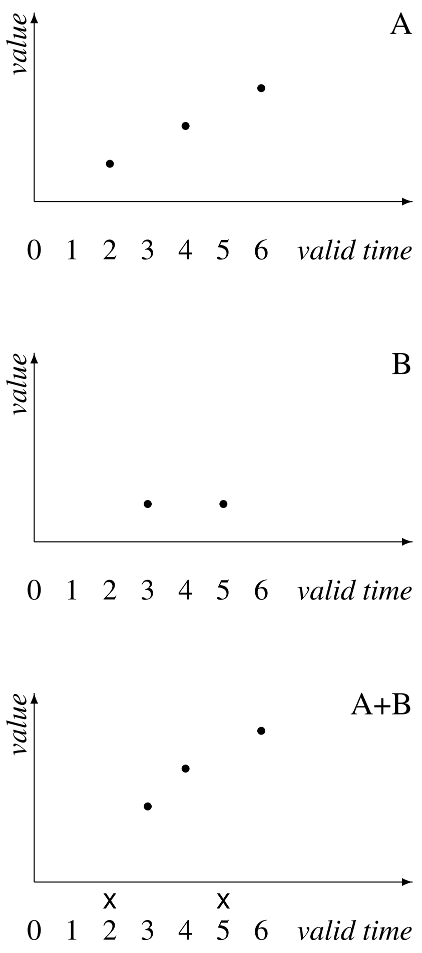


Figure 4.2: Timing of addition as an example for the reaction to missing values. The graph at the bottom shows the result while the graphs above show the input. Initially, no output is produced which can be interpreted as undefined. With valid time 2 one input produces the value 1. Since the other input is undefined, the addition yields *undefined* – shown as an x below the time axis. With valid time 3 the other input arrives, so a valid sum can be computed. With valid time 4 the trust period for channel B has expired with a new value being supplied, so the output is *undefined* again. Note that here the recalculation was triggered by time out rather than new input. With valid time 5 a value from channel B arrives (together with a value from A), so a new valid sum is calculated.

Input. These calculations take two or more numbers from different channels at the same time (FloatDataPointSet).

Output. The difference, quotient, root, exponent, or logarithm of the two inputs. If at least one input value is undefined, the output is undefined.

Timing. As for n-ary calculations.

4.2.1.3 Unary calculations

This section summarizes the behaviour of absolute value and sign.

Input. Both operations take one number as input (FloatDataPoint).

Output. The absolute value or sign of the input (FloatDataPoint). If the input is undefined, the output is so, too.

4.2.2 Date and time

This section describes miscellaneous utility functions related to the temporal dimension.

4.2.2.1 Current date and time

Some applications operate with time and date in addition to time annotated values. They also need the current time point – the *now* – as a reference. It is also defined in Asbru as a special value.

Input. None.

Output. The current date and time (TimeDataPoint). In Asbru, it is a value of type date, in the implementation it is an `int` just like the valid time of data points. Indeed, the valid time and the value have the same content. Still it is necessary to duplicate the value in order to make this data point a regular input to modules such as *add*.

Timing. Output is only produced when needed, as for *constant* above.

4.2.2.2 Current day of week

In some domains the day of the week is an important part of data abstraction. One example is diabetes, where you look for weekly patterns or distinguish measurements during the weekend from those during the week.

Input. None.

Output. The current day of the week as qualitative value of type *weekday* (new in Asbru 7.4).

Timing. Output is only produced when needed, as for *constant* above.

4.2.2.3 Valid time of measurement

Sometimes explicit reasoning about the temporal dimension is desired. Then the valid time – usually hidden in the background of the abstraction process – must become a *value* of data points. This is accomplished by this module.

Input. A data point.

Output. A data point where the integer *value* is equal to the valid time (DateDataPoint).

4.2.2.4 Duration

This element makes the durations of intervals of a steady qualitative value as values available.

Input. A qualitative data point.

Output. A data point where the integer *value* is equal to the difference between the valid time of the latest data point and the preceding one (TimeDataPoint).

4.2.2.5 Delaying data

In most cases one wants to have all the information as soon as possible, so artificial delays seem ridiculous. Still there are important applications for it, most when comparing several periods of time with each other, e.g., the average in the most recent minute with that of the minute before. While the change in average could be computed by connecting a ChangeModule to an AverageModule, such questions as "Was the average in the most recent minute twice as high as in the minute before?" can only be answered by connecting the average to a the module described here, multiplying this delayed average and comparing it against the most recent one (using a GreaterModule).

Input. Any data point.

Output. The same data point, with the delay added to the valid time.

Arguments. The time by which the input is delayed.

Timing. Output is produced whenever new input arrives. Therefore, the data points output can lie in the future which is not a problem for the management module, which stores them until it is time to produce that time slice.

4.2.3 Logical operations

The modules described in this section implement the Boolean operators, switching between different input channels, and the comparison of two values.

4.2.3.1 Logical conjunction

Input. Two or more Boolean values (IntDataPointSet).

Output. True if all inputs are true, false if at least one of the inputs is false, unknown otherwise, i.e., in cases in which there is at least one unknown input and all others are true.

4.2.3.2 Logical disjunction

Input. Two or more Boolean values (IntDataPointSet).

Output. True if at least one input is true, false all are false, unknown otherwise, i.e., in cases in which there is at least one unknown input and all others are false.

4.2.3.3 Logical XOR disjunction

Input. Two Boolean values (IntDataPointSet).

Output. True if both inputs are known and unequal, false if they are equal and unknown if at least one input is unknown.

4.2.3.4 Logical NOT

Input. One Boolean value (IntDataPoint).

Output. False if the input is true, true if it is false, unknown if it is unknown.

4.2.3.5 Switching between alternatives

There are situations in which the output of an abstraction step cannot be described as a logical or numeric function, but only in the form of conditions under which different forms of output are produced – similar to an if-then-else statement on programming languages.

Input. For each processing step, several values are input. They form pairs. For each pair, the second value is output if the first value is greater zero. Otherwise the next pair is examined. If the number of inputs is uneven, the last input is output if the first value of all pairs preceding this last number were not greater zero. Consider for example the tuple (a, b, c, d, e). If $a > 0$ then b is output. Otherwise, if $c > 0$ then d is output. If neither $a > 0$ nor $c > 0$ holds, then e is output. If the tuple would not contain e but be a quadruple, the undefined value would be output if neither $a > 0$ nor $c > 0$ holds.

Output. One of the inputs, selected as described above. Since the input is passed through, it is not restricted to data point, it can also contain sets of data points. This feature is important for the implementation of context selection, which is an feature specific to the Asbru language.

4.2.3.6 Comparison

Among other usages, the modules described here are used to implement the *value description* in the Asbru element *parameter-proposition*. Table 4.3 lists operators, Asbru syntax and module names.

Input. Two data points.

Output. A Boolean data point.

Operator	Asbru name	Module name
$<$	*less-than*	LessModule
$\leq$	*less-or-equal*	LessEqualModule
$>$	*greater-than*	GreaterModule
$\geq$	*greater-or-equal*	GreaterEqualModule
$=$	*equal*	EqualModule
$\neq$	*not-equal*	UnequalModule

Table 4.3: Comparison modules.

4.2.4 Miscellaneous abstractions

The miscellanea found in this section comprise the simple abstraction of qualitative values based on quantitative ones, a module to implement constants for use with comparisons and calculations, and a module mapping known values to the Boolean value *true* and unknown values to *false*.

4.2.4.1 Qualitative values based on numeric input

While the above algorithm provides a sophisticated answer to suppressing undesired oscillations, in many cases a simple solution is more suitable. This module simply maps its numeric (quantitative) input to qualitative values represented by integers using a range of limits.

Input. A numerical value plus a set of limits. These limits are not fixed, for each numerical input another set of limits can be supplied. This is necessary to implement Asbru's context dependent data abstraction feature which is described in detail in Section 4.7.7.4.

Output. An integer representing the qualitative value where 0 represents the value corresponding to the numerical range between the first and second limit.

4.2.4.2 Constant

Motivation. Some modules allow many degrees of freedom some of which are not always needed. In this cases the input can be feed from this module.

Input. None.

Output. The constant given in the argument.

Arguments. The constant.

Timing. Output is only produced when needed, i.e., when other modules producing input for the module to which this module is connected produce output. It is neither useful nor technically possible to connect the use a constant as sole input of another module. In such a case the output of the constant module would never be triggered.

4.2.4.3 Checking for existence

Motivation. There are Asbru elements querying whether a parameter has a know value at the current point in time or not. This can be implemented by this simple mapping module.

Input. One parameter of any type.

Output. True if the input is not *undefined*. False if it equals *undefined*.

4.3 Multiple sliding time windows for statistical analysis

The objective of the algorithms in this section is to use overlapping time windows to individually analyze fractions of the input data-stream. They are called sliding time windows because the interval under examination is slid along the time axis as time progresses.

The step width is independent of the size of the window. Therefore, the first can be adjusted to the granularity at which the input should be examined while the second solely depends on the size of the searched pattern.

Time windows of different size can be used in parallel. The logical combination of their output is often required to provide results of sufficient reliability.

In the Asgaard data abstraction unit, the creation of time windows is separated from the analysis of its content. The following first describes three different ways to create time windows (Section 4.3.1). Then, the analysis methods which can be applied on the content of time windows are described (Section 4.3.2). There are several simple feature extraction modules to make properties such as duration of an episode available to further abstractions. They are described in Section 4.3.3. Some of the abstractions based on time windows – most importantly linear regression – themselves produce complex results. Specialized feature extraction modules are applied to access single features in such cases. They are described in Section 4.3.4.

4.3.1 Types of time windows

There are three ways to specify the extent of time windows in the Asgaard data abstraction unit: (fixed) temporal extend, number of measurements, and episode.

4.3.1.1 Time windows based on time interval

The most intuitive way to specify the size of a time window is to give its size in units of time such as 1 hour. This means that the number of measurements in a time window can vary, if they are not available at a fixed rate without any gaps.

To ensure that missing data do not induce abstraction with a basis which is too weak, i.e., based on too few measurements, a minimum number of measurements can be given. If fewer measurements lie in the window, then the result is *undefined*, i.e., a gap in the abstracted values is inserted.

Input. A series of measurements from a single source (FloatDataPoint).

Output. A data point series object (FloatDataPointSeries). Undefined input values are skipped. Output can be sorted by value or transaction time. Note that first data point stored in the object need not lie inside the time window. It is still needed to define the value at the start of the time window. Compare figure 4.3.

Arguments. Size of the time window and frequency of output. Both are of Asbru type time. Plus a flag indicating whether sorting is by value or by transaction time.

Timing. The output is performed on a regular basis defined by the output frequency. To ease synchronization, the output is produced whenever the current time is a multiple of the output frequency. This means that after the start a much smaller set of measurements is output at the first moment of output.

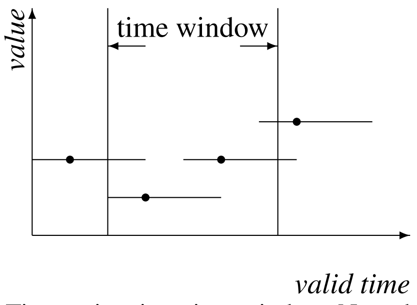


Figure 4.3: Time points in a time window. Note that the first and the last measurement in the picture both contribute to the values in the time window.

4.3.1.2 Time windows based on number of measurements

In some cases, one wants to summarize the previous n statements independent of the time at which they were taken. This of course imposes some vagueness at the interpretation of the result, but on the other hand ensures that exactly the expected number of measurements is the basis of the calculation.

To avoid too excessive growth to the time window's duration, a maximum duration (in time units) can be specified. If the previous n measurements lie further apart, the *undefined* value replaces this instance of time window to show that here no valid abstraction was possible.

Input and output. As above.

Arguments. Size and output frequency given as number of measurements. Flag indicating whether sorting is by value or by transaction time.

Timing. Output is produced whenever the given count of input measurements is read. There is no synchronization in terms of actual time intervals.

4.3.1.3 Time windows based on episodes

Sometimes for each occurrence of a certain episode one wants to calculate a certain aggregation the measurements during this episode. Suppose, for example, we want to compare the standard deviation of the heart rate during different hypoxic episodes. An hypoxic episode is defined as SpO_2 lower then 80 for at least 4 seconds. Start and end of each such episode form a time window. These time windows are of extremely varying size and they are not at all evenly distributed over the total duration of the measurement. Therefore, regular time windows are not applicable at all. Furthermore, we want to precisely cut off the time window at start and end of each episode to avoid the distortion of the results by values outside the episode.

The episode based time window is designed to meet these requirements. It takes a stream of episodes produced, e.g., by matching a parameter proposition or a generalized pattern description, and a second, independent input which supplies the values to be grouped in time windows. For each episode one time window is output, starting at the start of the episode, ending at the end of the episode, and containing all values from the second input which fall into this temporal interval.

Input. A series of measurements from a single source (FloatDataPoint) plus episodes (EpisodeDataPoint).

Output. As above.

Argument. A flag indicating whether sorting is by value or by transaction time.

Timing. Output is produced whenever an episode is complete, i.e., both start and end are known and end already passed. At this point in time a window with all measurements recorded during the episode is output.

4.3.2 Analyzing time windows

The stream of time windows generated by one of the three modules described in the previous subsection is feed into one or more of the modules described in this section. Only the combination of both abstraction steps provides output meaningful for the user.

In all cases, output is produced whenever new input arrives. This means that if the input comes from a TimeWindowModule, output will be produced on a regular basis. If input comes from a MeasurementSeriesModule, it will only be as regular as the input series. If input comes from a EpisodeTimeWindowModule, output is only produced when an episode is found, i.e., rather irregularly.

4.3.2.1 Parameterless statistical measures

The standard abstractions median, minimum, maximum, average and standard deviation are all based solely on the content of a time window without further parameters. All these measures ignore the temporal dimension of the input.

Input. Sorted data point series produced by a TimeWindowModule, MeasurementSeries-Module, or EpisodeTimeWindowModule.

Output. The median, minimum, maximum, average, or standard deviation of the series.

4.3.2.2 Change

Analyzing the change of a value over time is a cheap and often sufficient way of determining the trend in a measurement series, provided that the input is not too noisy. For a moderate, constant amount of noise, averaging a small time window and observing the change between these averages is an attractive alternative to calculating the slope of a linear regression (Section 4.3.3.2 and 4.3.4.1).

This module simple calculates the difference between the first and the last point in the time window. This means that most of the values in the time window are ignored. Still, this is an intended behaviour for slowly changing values where the assumption, that the values in between form a monotonous connection of the first and last value is justified.

Input. Either a time window or a series of data points.

Output. The difference between the first and the last value with the valid time of the last.

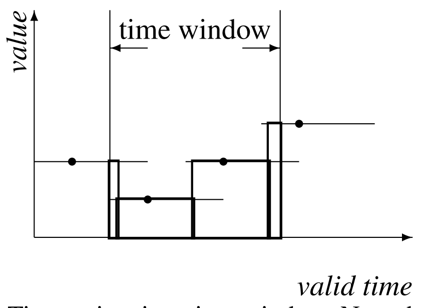


Figure 4.4: Time points in a time window. Note that the first and the last measurement in the picture both contribute to the values in the time window.

4.3.2.3 Time-oriented average of data-point series

This module does not only average the values of the data points but instead adds the areas of all rectangles defined by the value of each measurement and its valid time interval as shown in figure 4.4.

Input. Unsorted data-point series produced by one of the three time window producing modules.

Output. The average given as number of the same type (at the Asbru level) as the input (FloatDataPoint). The average is not calculated from the values alone, but each value is weighted with the duration of its validity.

4.3.2.4 Centiles

The value of an X-%-centile is that value in the time window for which X % of the values in the time window are smaller. The described implementation of this abstraction allows dynamic changes of the threshold X to adapt to changing context.

Input. A sorted series of data points and a percentage.

Output. The value in the series of data points for which the given percentage of data points in the series are smaller then itself.

4.3.3 Accessing time window properties

This subsection describes simple abstractions which are similar in their input – the content of a time window – but dissimilar in their output. While the first group produces numeric time values, the second generates regression lines which themselves are complex objects; their properties are accessed by modules described in Section 4.3.4.

4.3.3.1 Start, end, and duration

In some cases, explicit calculations based on the start or ending time of each of the set of data points which actually form the time window are necessary. The start is defined as the valid time of the first measurement in the time window. The end is the valid time of the last measurement. The duration is the difference between these two values.

Note that even for the time window defined by its temporal extend (TimeWindow-Module) the actual time span between the first and last value can be a varying entity for irregular measurements.

Input. A series of data points produced by one of the three time window producing abstraction modules.

Output. A numeric value of Asbru type date in the first two cases, and of Asbru type time in the third case reflecting the first valid time, last valid time, and difference between them respectively.

4.3.3.2 Linear regression

All of the above abstraction based on time windows ignore the temporal order of the measurements. However, treating the measurements in a time window in a two dimensional way instead of ignoring the temporal dimension is important if the measurements show an increase or decrease over time. In this case, the line resulting from a linear regression (approximation) of the content of the time window is not horizontal.

This has two consequences: First, the ascent of the regression line is a good estimate of the trend of the measurements, and second the standard deviation and other measures for the deviation of single data points from the regression line are smaller and more precise then for a one-dimensional analysis.

A side product of this analysis is the temporal dimension of the centre of the distribution. For regular measurements, this is the middle of the interval. If there is a gap in the measurements lying not in the middle, then the centre lies nearer to the majority of measurements. This image shows that the centre represents the "centre of gravity" of all measurement in the time window which could be used in calculation trust in the result. However, in online settings this advantage is completely lost by the fact that the result of the analysis is not known before the end of the interval.

Input. A series of data points produced by the modules in Section 4.3.1.

Output. A linear regression line object (RegressionLineDataPoint). It contains the centre (time and value), the standard deviation, the standard error, the slope and the start and end of the time window for which the calculation was done.

4.3.4 Properties and abstractions of linear regression

This section comprises a group of simple modules which output numeric properties of regression lines on the one hand, and the intersection of a regression line and a horizontal line demarking an alarm value, on the other hand.

4.3.4.1 Extraction of details of the regression line

The following features of a regression line can be extracted. All of them are output in the form of a FloatDataPoint.

slope. The ascent or slope of the regression line, i.e., the trend of the time window from which the regression was calculated.

standard deviation. The standard deviation is the calculated as the root of the sum of squared distances between the data points and the regression line.

standard error. The standard error is calculated by multiplying the standard deviation by $\frac{\sqrt{m}}{\sqrt{n}}$ where m is the maximal count of time points in the time window and n is the number of valid data points in the time window.

end point. The end point of the regression line is formed by the valid time of the last measurement and the y-coordinate of the regression line at this position.

centre. The centre is the point in the two-dimensional plane formed by value and time axis, for which the squared distance to all data points is at its minimum.

Input. A regression line.

Output. As described above.

Arguments. A flag indication whether the time of the *centre* or the *end* of the regression line should form the valid time of the data point.

4.3.4.2 Time to alarm

In online monitoring, the time until a certain threshold is reached if the current trend continues is an interesting parameter. This is calculated as the intersection of the regression line with a horizontal line defined by the threshold.

Input. A regression line and a threshold.

Output. The time left until the threshold is reached. I.e., the difference between the (continuation of the) regression line and the horizontal line defined by the threshold on the one side and the end point of the regression line on the other side. If the trend leads away from the threshold, the result is *infinite*. If the regression line crosses the threshold before the end of the regression line is beyond the threshold, the result is negative.

Arguments. A flag indicating whether the threshold is an upper limit or lower limit. If the starting end of the regression line is below the threshold and the end point is above, and if the threshold is an upper limit, then the result is the negative temporal distance between the end point and the intersection. If in the same example the threshold is a lower limit, the result is infinite.

4.4 Coping with noisy data

The previous section described a set of abstraction modules on a technical level. This section takes an application focus and describes how the above modules can be combined to cope with noisy data in different ways. This is followed by the introduction to more complex abstractions in Section 4.4.4.

Data in the medical domain is often distorted by single errors and noise originating from technical limitations in the monitoring process and circumstances in the treatment process such as interventions of the care personal. In addition, data may temporarily be missing. There are several possible reactions to this, ranging from simple to complex.

There are several simple rules to detect errors in data, which can be defined for most types of input. Examples are a maximum and a minimum for the value (e.g., it is not possible for a living person to have a body temperature of 30^oC). Nearly as simple to find is a maximum for the change in this value.

The simplest reaction to finding an error is to remove the data point. This causes a gap in the stream of measurements which has undesired effects. Another option is to replace the faulty value by some estimate. Since this risky, it needs control by explicit rules defining the estimate in many cases. Section 4.4.2 shows some examples based on previous work in this field.

The most attractive solution is the design of abstraction algorithms which perform sufficiently even on noisy input data. This of course is not possible for the general case, but it is for specific cases. The fundamental precondition for success is integration of domain knowledge about the interpretation of the results of abstraction. E.g., it may be safer to assume a value lower than it actually is in one case, while in another case it is safer to assume a higher value. This depends not only on the information behind the measured value, but only on the context, which may change over time.

Descriptive statistics offer a set of stable abstractions from varying input data. Section 4.4.3 describes the means available in our system and how to combine them for more sophisticated quantitative abstractions. While simple measures such as exclusion of outliers may be sufficient for a group of tasks, there is a considerable amount of situations in which only specific combinations of different methods arrive at sufficiently clean data without loss of too much information.

Medical knowledge is often expressed using qualitative concepts rather then quantitative ones. Abstracting such concepts directly from noisy quantitative input reduces the distortion of the input through intermediate abstractions such as averaging, but handling varying amounts of noise is not trivial. Section 4.4.4 shows some solutions to this problem.

4.4.1 Error detection

In this section, the detection of errors using fixed and dynamically changing limits as well as arbitrary rules is described.

The basic reaction to errors is the removal of the faulty value. The resulting gap in the input need not be necessary. In some cases, previous measurements allow the approximation of the real value. In this section, only the solution of taking the previous, correct measurement is used. See Section 4.4.2 for more complex estimates.

4.4.1.1 Static limit check

A considerable fraction of input errors can be detected by simple checks. This module compares the input against a given minimum and maximum. It also compares the change since the last measurement against the given minimum and maximum. If one of these four checks fails, then the reaction depends of the flag *mark gap* given as argument. If it is set to *true* on the implementation level or *yes* on the Asbru level, the undefined data point is output, otherwise nothing is output.

The consequence of not outputting the undefined data point is that the parameter may still be considered valid and having the last correct value, as long as the trust period of this measurement is not expired.

If the value does not exceed any of the limits, it is output unaltered.

If one of the limits is not defined, the corresponding value can be set to positive or negative infinity to disable the check.

Input. The following input values are of type FloatDataPoint.

- The measurement to be examined.
- Minimum of the value.
- Maximum of the value.
- Maximum of the difference between the previous value and this value. It is given per time step, i.e., if the atomic time step is 1 second and 2 seconds passed since the previous measurement, then the double change is allowed without exceeding the limit.
- Minimum of the difference. This is identical to the maximum of decrease with a negative sign.

Output. Same as input, except if one or more limits are exceeded. Then a data point with the undefined value is output, if *mark gap* is *true*.

Arguments. The flag *mark gap* specifying whether a faulty value should be replaced by the undefined value or simply be ignored.

4.4.1.2 Dynamic limit check

The above module method is not capable to react to changing amount of noise in the data. If there are wide oscillations, a large amount of the input will be discarded, while no outlier will be removed if the oscillations are small.

In many cases, a pragmatic approach is necessary, such as the elimination of a certain percentage of outliers, independent of their actual value and the distribution of the remaining value. Since the LimitModule described above allows for dynamically changing limits, such a solution can be implemented by replacing the static limits by arbitrary abstractions.

Figure 4.5 shows the definition of 10 % centiles as the limit of a value. This means that the resulting value, *X-cleaned* contains the central 80 % values of the values in the raw data *X*. In the example, the limits are calculated for the previous minute.

```
parameter-def name=”X-raw” type=”distance”
    raw-data-def mode=”automatic”
parameter-def name=”X-cleaned” type=”distance”
    limit-def
        maximum-value
            centile-def
                limit
                    numerical-constant value=”90” unit=”%”
                source
                    parameter-ref name=”X-previous-minute”
        minimum-value
            centile-def
                limit
                    numerical-constant value=”10” unit=”%”
                source
                    parameter-ref name=”X-previous-minute”
        source
            parameter-ref name=”X-raw”
parameter-def name=”X-previous-minute” type=”time-window”
    time-window-def
        window-length
            numerical-constant value=”1” unit=”min”
        step-width
            numerical-constant value=”1” unit=”sec”
        source
            parameter-ref name=”X-raw”
```

Figure 4.5: Dynamic limit checks. *X-cleaned* is X-raw without any measurements lower or higher then 90 % of the measurements in the previous minute.

4.4.1.3 Complex rules for the dynamic definition of erroneous measurements

The application if the limit module as described above is limited to cases in which numeric limits for a certain numeric value are defined. If the validity is defined by more complex rules, a *logical-dependency-def* is needed to implement them.

The basic design is to implement the rules which define the validity of the input as a combination of abstraction steps and to gate it through the mentioned logical dependency module. There are two ways to react to invalid input. Either a *undefined* value is output, or no value at all is output. In the second case, the value of the previous (valid) measurement may be used until the trust period is over. This is an important option in cases where it is important to avoid gaps in the data and where it is safe to assume that suspicious values result from transient measuring faults rather then the development of a new – potentially alarming – trend.

In the following example, *A-checked* is abstracted from *A-raw* by eliminating any values for which both the value itself and the absolute value of change lie above the average for *A-checked* for the previous minute. To this end, the value of *A-raw* is passed on as *A-checked* if and only if *A-is-ok* is true. The Boolean value of *A-is-ok* is a true if either *A-raw* is lower or equal than the average of *A-checked* or if the change is lower or equal than the average change of *A-checked*.

If *A-is-ok* is not true, no output of *A-checked* is produced. This means that the previous measurement is only invalidated through the end of the trust period which is defined as 5 hours.

4.4.2 Rule-based repair of data

Leaving a gap in the stream of information should only be the last resort. Often, assumptions about the real value of the missing parameter can be made for a certain time after the last valid measurement. The function to compute replacements for the missing values is always highly domain dependent. Therefore, the Asgaard data abstraction framework does not provide default extrapolation modules, but allows the definition of any estimation as abstractions based on arbitrary previous measurements (potentially from different channels).

In the example in Figures 4.7 and 4.8 gaps of up to one minute in the raw data (*raw*) are closed by the following function: The gradient (*estimated-change*) is supposed to approach zero in 30 equal steps during the first 30 seconds, starting with the last measured gradient (*measured-change*). For the remaining 30 seconds, the value is estimated to be equal.

In the following equation, C_i is the change or gradient of the function i seconds after the last measurement, and V_i is the value of the estimation function i seconds after the last measurement.

$$\begin{array}{rcll} V_0 & = & M & \\ C_i & = & \frac{C \cdot i}{30} & \text{if } i \leq 30 \\ & & 0 & \text{otherwise} \\ V_i & = & V_{i-1} + C_i & \end{array}$$

```
plan-library
  domain-defs
    domain name="complex-limit-check"
      parameter-group
        parameter-def name="A-raw" type="weight"
          raw-data-def unit="g" mode="manual"
        parameter-def name="A-is-ok" type="boolean"
          logical-combination-def operator="or"
            comparison-def operator="less-or-equal"
              left-hand-parameter
                parameter-ref name="A-raw"
              right-hand-parameter
                average-def
                  interval
                    numerical-constant value="1" unit="h"
                  source
                    parameter-ref name="A-checked"
            comparison-def operator="less-or-equal"
              left-hand-parameter
                calculation-def operator="absolute-value"
                  change-def
                    interval
                      numerical-constant value="2"
                    source
                      parameter-ref name="A-raw"
              right-hand-parameter
                average-def
                  interval
                    numerical-constant value="1" unit="h"
                  source
                    calculation-def operator="absolute-value"
                      change-def
                        interval
                          numerical-constant value="2"
                        source
                          parameter-ref name="A-checked"
        parameter-def name="A-checked" type="weight"
          trust-period
            numerical-constant value="5" unit="h"
          logical-dependency-def
            if
              parameter-ref name="A-is-ok"
            then
              parameter-ref name="A-raw"
```

Figure 4.6: Checking A against its average and the average of its change.

```
plan-library
   domain-defs
      domain name=”repair”
         parameter-group
            parameter-def name=”raw” type=”volume”
               raw-data-def mode=”automatic” unit=”ml”
            parameter-def name=”time-since-last-measurement” type=”time”
               calculation-def operator=”subtract”
                  now
                  valid-time-def
                     parameter-ref name=”raw”
            parameter-def name=”measured-change” type=”volume”
               change-def
                  interval
                     numerical-constant value=”1”
                  source
                     parameter-ref name=”raw”
            parameter-def name=”estimated-change” type=”volume”
               logical-dependency-def
                  if
                     comparison-def operator=”less-or-equal”
                        left-hand-parameter
                           parameter-ref name=”time-since-last-measurement”
                        right-hand-parameter
                           numerical-constant value=”30” unit=”sec”
                  then
                     calculation-def operator=”divide”
                        parameter-ref name=”measured-change”
                        numerical-constant value=”30”
                  default
                     numerical-constant value=”0”
```

Figure 4.7: Sample repair function defined in Asbru, part 1. The change is assumed to approach zero in equal steps during the first 30 seconds of the gap. The estimate is valid for the first 60 seconds only.

```
parameter-def name=”estimated-value” type=”volume”
  logical-dependency-def
    if
      comparison-def operator=”less-or-equal”
        left-hand-parameter
          parameter-ref name=”time-since-last-measurement”
        right-hand-parameter
          numerical-constant value=”60” unit=”sec”
    then
      calculation-def operator=”add”
        delay-def
          delay
            numerical-constant unit=”sec” value=”1”
          source
            parameter-ref name=”estimated-value”
        parameter-ref name=”estimated-change”
parameter-def name=”resulting-value” type=”volume”
  logical-dependency-def
    if
      parameter-proposition parameter-name=”raw”
        is-known-parameter
        context
          any
        time-annotation
          now
    then
      parameter-ref name=”raw”
    default
      parameter-ref name=”estimated-value”
```

Figure 4.8: Sample repair function defined in Asbru, part 2. If there is a valid measurement, it is used as the result. Otherwise, the estimate is used, which itself is valid for the first 60 seconds only.

4.4.3 Stable quantitative abstractions

Standard statistical analysis provides an easy to communicate way to abstract potentially noisy data. The result is a quantitative value, i.e., a floating point number.

4.4.3.1 Average and median

Average and Median are popular means to estimate the centre of a distribution. Using a sliding time window (Section 4.3) and the modules described in Section 4.3.2.1 they can be computed for arbitrary time periods which are advanced in time steps of choice.

The results for various time windows can be freely combined. This allows to "fail to the safe side" in those cases where the safe side is known. E.g., if it is unsafe to assume a certain value too high, considering the minimum of the median for the previous minute and the previous three minute interval should be safe.

The median can also be used to compute a replacement for missing values. E.g., if the most recent value exceeds the median by Y %, that value is used instead of the too high measurement. Such an abstraction could be considered a repair function, if used by all further abstractions. It can also be interpreted as special purpose abstraction if other abstractions from the same noisy input use different assumptions or restrictions.

4.4.3.2 Centiles

To increase the level of safety, the median may be replaced by a centile. E.g., suppose it is alarming if $A > B$ and both are oscillating and we want to avoid false alarms. Comparing the median or average of both parameters for the previous minute[1] will reduce false alarms but introduce the danger that the alarming conditions exists for several seconds and that these measurements are correct. Reducing the time window size to a few seconds will reduce this danger, but bring back false alarms to some extend.

Replacing the median of A by its 60 % centile means that an alarm is triggered if only 40 % of the measurements of A in the previous minute are bigger then the median of B. This way we gain sensitivity while retaining the size of the time window. After replacing the median of B for its 40 % centile, the absence of an alarm means that 60 % of the measurements for A are smaller than 60 % of the measurements for B. For the medians, this guarantee referred to only 50 %.

The decision, which of the solutions is the most appropriate greatly depends on the possibilities of knowledge acquisition. Clinical knowledge may be available in a large range of different forms. It is important for the knowledge engineer to have a equally diverse set of abstractions at his hands to meet the demands from practice.

4.4.4 Abstraction of qualitative values from noisy quantitative input

Qualitative concepts such as *high* and *low* values play an important role in medical knowledge. They are abstracted from numerical values by comparing the input against certain limits[2]. The qualitative abstractions are should be stable, i.e., that they do not

[1]It is assumed that new measurements arrive once per second.

[2]These limits are context dependent. Here, I focus on the problems caused by noise in the input.

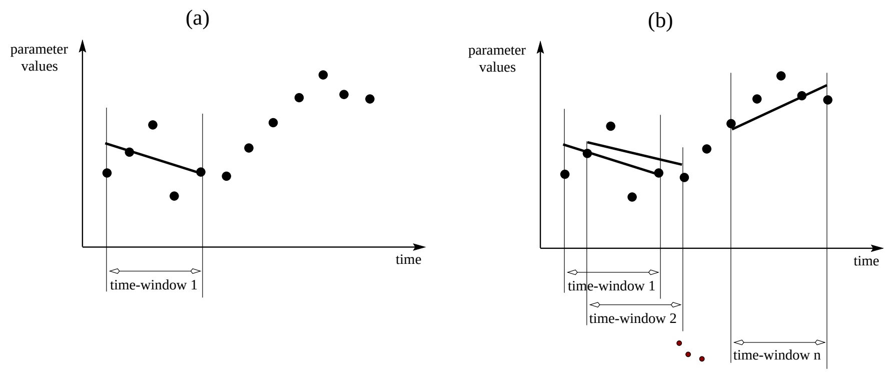


Figure 4.9: For each instance of a sliding time window of fixed size a linear regression is calculated. (a) shows a single time window, (b) the whole series [94].

change between different values unless necessary. Oscillations in the input may cause such undesired switches between different qualitative values.

A standard solution to this problem is the introduction of a threshold around each limit. However, the bigger the threshold, the bigger the delay in the abstraction. I.e., for a slowly and steadily increasing value, we would like to change to the next qualitative region as soon as the input value exceeds the limit. At the same time, for noisy data, we want to delay the change of the qualitative value, and once it changed, we want to maintain it even if the input value oscillates back to the previous region for insignificant fractions of the observation time window.

Static thresholds cannot help here. Instead, the threshold must be adapted to the amount of noise continuously.

The abstractions described below were developed and evaluated for the domain of controlling oxygen supply in a neonatal intensive care unit. However, they can be used in any other setting where steady qualitative abstractions are derived from input data of varying quality.

4.4.4.1 The Spread

The basic answer to representing oscillations of varying intensity is to take the following steps.

1. Find a centre value for the measurements in each instance of the sliding time window.

2. Calculate a measure of deviation within this time window.

3. Plot it up and down from the centre found in step 1.

4. Connect all upper ends of the resulting bars, and all lower ends.

The resulting band of varying width is called *Spread*.

The first design is described in [94] from which figures 4.9 to 4.11 are taken.

The centre value is the centre of a regression line laid through the data point in the time window. This centre will however carry the time stamp of the end of the window

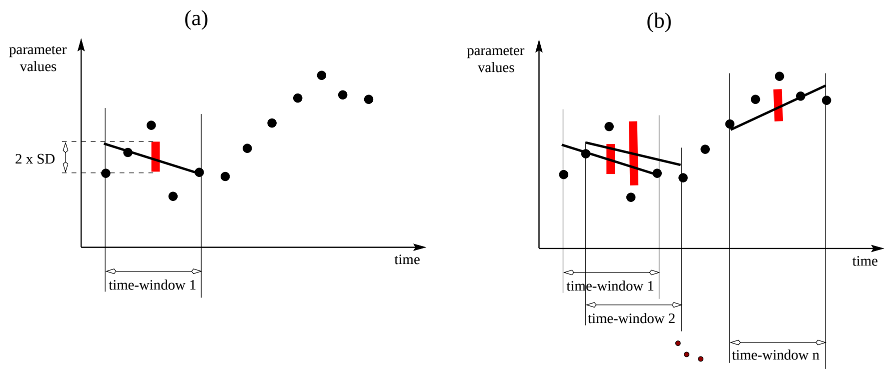


Figure 4.10: The measure of deviation is plotted up and down from the centre of each line [94].

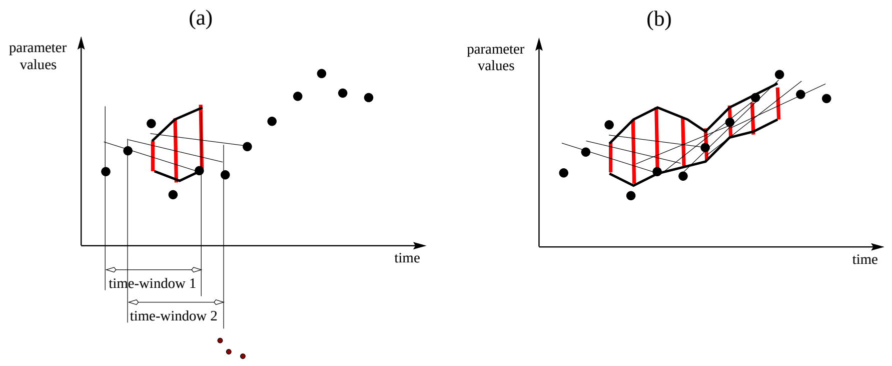


Figure 4.11: All upper end and all lower ends are connected to yield a band of varying width, called the *Spread* [94].

in real-time monitoring, since only at the end of the time window this calculation can be performed.

The measure of deviation was the standard deviation or standard error in the original version. However, the input data – oxygen saturation measured by pulsoximetry on neonates – proved to be asymmetric and the standard deviation overly exaggerated outliers, from the perspective of the application at hand (finding a stable estimate of the oxygen level to devise necessary changes in the oxygen supply).

Therefore, I replaced the standard deviation by centiles. They are calculated of values above and below median independently, and the value of outliers does not influence them, only the value of the majority of data points does.

Input. A linear regression line object (RegressionLineDataPoint, compare Section 4.3.3.2).

Output. A twin data point representing the limits resulting from this regression line. Depending on the variant of the module, the distance between the two data points and the center of the regression line is the standard error, the standard deviation, of a centile.

Arguments. For the centile, the argument holds the percentage of the cut-off, i.e., 10 for a 10% centile.

Timing. Output is produced whenever new input arrives. Note that the valid time is that of the end of the regression line.

4.4.4.2 Spread-based qualitative values

Based on the above described group of abstractions, a qualitative value is generated in a conservative way by only changing to the next value if the whole Spread changes to a new qualitative region.

Input. Two data points (FloatDataPointSet) either produced by two different input channels or by a channel producing pairs of data points such as the *Spread*.

Output. A qualitative value computed as described in the next paragraph.

Arguments. The value of the normal region and a flag *memory-on* indicating whether the starting point of assigning the new state should be either the previous state or the normal region. In both cases, the qualitative value is decreased as long as the upper limit of the Spread is lower then the lower limit of the current qualitative region. And it is increased as long as the lower limit of the Spread is higher then the upper limit of the Spread. Compare Table 4.4.

upper limit of Spread	25
lower limit of Spread	15

Qualitative region	from	to
very low	-10	0
low	0	10
medium	10	20
high	20	30
very high	30	40

Base value	Resulting value
very low	medium
low	medium
medium	medium
high	high
very high	high

Table 4.4: Qualitative value computed for a Spread depending on different starting values. The base value is the previous resulting value if *memory-on* is true or the normal value otherwise.

4.5 Online-algorithms for monitoring temporal patterns

In real-world applications such as medicine, plan execution must be synchronized with the environment, i.e., the patient's health condition. One of the main features of the Asbru language is to provide a strong bridge between the observation of the environment and the plan execution. This is achieved by changing the plan states if and only if certain conditions are fulfilled.

However, until the introduction of the solutions described in this chapter, only implementations which relied on repeated database queries were available [97]. They were not suitable for high-frequency domains.

In the following, I take a bottom-up approach to describe them. First some deeper details of the concepts in the Asbru language are discussed which affect the design of monitoring algorithms. Then each of the major issues is discussed in detail. These are:

- Parameter proposition (Subsection 4.5.2 on page 98)
  - with fixed reference point
  - with repeated reference point
  - with reference point now
- Plan state constraint (Subsection 4.5.3 on page 116)
- Temporal constraint (Subsection 4.5.4 on page 116)
- Boolean combination of temporal patterns (Subsection 4.5.5 on page 121)
- Count constraint (Subsection 4.5.6 on page 131)

Section 4.1 describes how the output of these algorithms is integrated and used by other modules. Section 4.7 describes how the relevant language elements map to the abstraction modules described here or to simpler ones which are described in Section 4.2.

4.5.1 Related language features of Asbru

The above mentioned conditions in Asbru are based on *temporal patterns* containing *parameter propositions* which again contain *time annotations*. The subsections 4.5.1.1

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

Table 4.5: Defaults for unspecified parts of a time annotation.

to 4.5.1.4 describe these basic concepts – time annotation, parameter proposition, and temporal pattern. In subsections 4.5.2 to 4.5.7, concrete algorithms for the monitoring process are given.

4.5.1.1 Time annotation

A time annotation describes an interval of time in a flexible way. It specifies an interval within which the starting point must lie and another one for the end point. In addition, it specifies minimum and maximum of the duration of the interval. Using three intervals to constrain start, end, and duration of an interval was first proposed by Rit [112]. In Asbru, all 6 constraints are defined based on a reference point. Using different reference points in different time annotations allows for different time lines in the monitoring process. In detail, a time annotation consists of

Reference point (RP): The time point, to which the following four values (called *shifts*) are offsets. Examples for reference points are the birth of the patient, *now* (the current time point), or some event like the start of a plan.

Earliest starting shift (ESS): The earliest point in time (relative to the reference point), at which a fitting interval may start.

Latest starting shift (LSS): The latest point in time (relative to the reference point), at which a fitting interval may start.

Earliest finishing shift (EFS): The earliest point in time (relative to the reference point), at which a fitting interval may end.

Latest finishing shift (LFS): The latest point in time (relative to the reference point), at which a fitting interval may end.

Minimum duration (MinDu): The shortest time span of an interval to be a fitting interval.

Maximum duration (MaxDu): The longest time span of an interval to be a fitting interval.

Each of these except the reference point may be undefined. In this case, the defaults shown in Table 4.5 are used.

An illustrative example. Something happens on the second day of life.

- Reference point is the birth of the patient.
- If we want to express that the interval is fully contained in this day but do not care about the duration, then we demand that ESS=1 day, LFS=2 days.
- To express that it happens the whole day, but we do not care when it started and ends, we give LSS=1 day and EFS=2 days, but leave ESS and LFS undefined.
- If we demand that the interval spans exactly the 2$^{\text{nd}}$ day, then we define all four spans as given above. Note that in this case, the chance to find a fitting interval is very small, since there is no flexibility both at start and end of the interval and thus an interval must last exactly 24 hours and start exactly at midnight to match the time annotation.
- In practice the shifts explicitly define intervals for start and end during which they are considered to be acceptable near the intended time points. So we could define ESS=23 hours, LSS=25 hours, EFS=47 hours, LFS=49 hours to demand that the interval spans the second day of life and not more, but we allow 1 hour tolerance in both directions on both ends.
- Now the interval will last between 22 and 26 hours. If so much uncertainty is not acceptable, MinDu=23 hours and MaxDu=25 hours can be defined to additionally limit the duration. Doing so, we arrive at a quite complex notion of an interval which spans quite exactly the second day of life and which allows start, end, and duration to deviate from the ideal by 1 hour in both directions.

A time annotation can be illegal or abnormal. An illegal time annotation is one which cannot match any interval. An abnormal time annotation can match intervals but is specified in a suboptimal way. These checks can be performed as soon as all values are known. This is the time at which the execution unit evaluates the time annotation. Only if all values are constants, the checks can be performed before starting plan execution.

Illegal time annotations. A time annotation is illegal if one of the following rules are violated.

$$\begin{array}{rcl}
ESS & \leq & LSS \\
EFS & \leq & LFS \\
ESS & \leq & EFS \\
LSS & \leq & LFS \\
MinDu & \leq & MaxDu \\
EFS - LSS & \leq & MaxDu \\
MinDu & \leq & LFS - ESS
\end{array}$$

While the first five simply define that the end must follow the start and the minimum must not be greater than the maximum, the last two are more complex. If $EFS - LSS > MaxDu$ then even the smallest possible interval – starting the latest and finishing the earliest is longer than MaxDu and therefore not admissible. Similarly, if $MinDu > LFS - ESS$ then even the longest interval formed by the earliest starting point and the latest finishing point is longer then MinDu.

Annormal time annotations. A time annotation is annormal if one of the following rules are violated.

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

If the first rule is violated, then MinDu could as well be undefined, since it is smaller than any interval allowed by the shifts ($EFS - LSS$ is the shortest one). The same hold for MaxDu if it is bigger then the longest interval stretching from ESS to LFS.

If $MinDu > EFS - ESS$ then an interval starting at ESS must end some time after EFS (at $ESS + MinDu$) so EFS could be increased to this value either MinDu or ESS should be decreased. The same holds for the latest finishing and starting shifts.

Similarly, if $EFS - ESS > MaxDu$ then an interval starting at ESS must end before EFS or exceed MaxDu. The same holds for $LFS - LSS$.

Note that these cases only describe suboptimal formalization of the time annotation. They are still valid and can occur in practice, especially in those cases, in which the values used in the time annotation are supplied by variables which are changed in a complex way during plan execution.

Types of reference points. The reference point in the time annotation can either be

- a single, fixed time point (possible only found at runtime, e.g., the start of a certain plan);
- a repeated time point, e.g., a change of a parameter to a certain value which can occur more then once;
- the moving time point now which designates the valid time the data abstraction unit is currently processing.

There is a fundamental difference in the monitoring process between these cases for time annotations in which at least one of the four shifts are defined. If only MinDu and/or MaxDu are specified, the reference point is arbitrary. These distinctions are discussed in the next subsection, since they are strongly connected to monitoring parameter propositions.

4.5.1.2 Parameter proposition

The heart of environment monitoring, i.e., the synchronization of plan execution with world state is the parameter proposition. It is used in the conditions governing plan state transitions, in the Asbru parameter definition *boolean-def*, and in the abstraction module *parameter proposition*.

In Asbru 6.5 it was only used in temporal patterns which again were only contained in conditions of plans. Permitting is usage in abstraction statements in Asbru, too, increases the efficiency of the monitoring process as the outcome of monitoring a certain parameter proposition can be used by more than one condition.

A parameter proposition is fulfilled, if a certain parameter has a certain value for a period of time described by a *time annotation* in a certain context.

A parameter proposition consists of the following.

Parameter name: name of the parameter referred to, e.g., *X* or *fever*.

Value description: either a value or a value range describing the permitted values of the parameter, e.g., *equal to 5*, or *ranging from 3 to 5*, or qualitative values such as *high*.

Time annotation: a described in the preceding section.

Context: A set of qualitative variables and their allowed values, e.g., *ventilation mode equals CPAP*.

Starting with Asbru version 7.2, value description, context, and time annotation can contain variables in Asbru. Therefore, the actual values passed on to the data abstraction unit are only decided during plan execution. For the issue of data abstraction, the only consequence lies in the fact that it is not possible to monitor the status of a parameter proposition in the plan library from program start, and that the same parameter proposition in the plan library will be represented by different instances of the parameter proposition module in the data abstraction unit depending on the changing values of variables.

Depending on its usage, the parameter proposition has one of two types of result:

- If used in a condition, it is *first not yet fulfilled*. If the first fitting interval is found, the parameter proposition (and the condition containing it) is *fulfilled*. Under certain circumstances it is also possible to decide, that a particular parameter proposition will be *never fulfilled*, i.e., there cannot be a fitting interval in the future (and there was none in the past. The implications of such a finding are discussed in detail in Section 4.5.1.4.

- If used as an abstraction step, the result of applying a parameter proposition is a series of fitting intervals. Note that a single parameter propositions can match more then one interval (e.g., the times of fever for more then one day in your life).

Abstracting from the value matching process, we define:

Positive flank (PF): A point in time after which the parameter has a value prescribed in the value description and before which it does not. In addition, the context described in the parameter proposition must be given, otherwise the parameter is ignored.

Negative flank (NF): The counterpart of a positive flank. It can be caused by either the value of the parameter or the context not meeting the description in the parameter proposition anymore. This means that if the parameter still has the correct value but the context changes and the new context does not fit the context description in the parameter proposition, the effect is the same as if the parameter value would have changed and does not meet the value description anymore.

Fitting interval: The interval formed by a positive and the consecutive negative flank which matches the temporal constraints of the time annotation.

In the following, we will focus on PF and NF, treating the input as Boolean and leaving the comparison of the parameter to the value description and the context to its specification to other modules of the abstraction system, as it is described in detail in Section 4.7.

If a time annotation with a fixed reference point is fulfilled, it will remain so forever. E.g., if the value for total serum bilirubin (TSB) is high on the second day of life, this fact can never change in the future. In contrast, a time annotation with reference point now can end being fulfilled. E.g., for a parameter proposition stating that the interval of high fever must start 4 hours before now and fever changes from normal to high now, then this parameter proposition is fulfilled for the current time being, but it is clear that it will not be fulfilled in 4 hours and 1 minute, since then the start of this episode of high fever will is past 4 hours from the then current time.

On the other hand, for a time annotation based on now there retains always a chance to be fulfilled in the future, since the reference point is forever changing and new intervals can be matched. In contrast, for a time annotation with a fixed time point and some of the shifts defined, we can find a time point after which it is sure that the parameter proposition will never more be fulfilled. E.g., a parameter proposition describing high TSB on the second day of life will never more be fulfilled after the end of that day, if it was not fulfilled on that day.

A time annotation with a repeated reference shares properties with both other cases. On the one hand, it is the same as a set of time annotations with fixed reference points – each instance of the repeated reference point founds one time annotation with this reference point as its single, fixed reference. On the other hand, since there will always be new instances of the repeated time point in the future[3], monitoring never completes.

The consequences of knowing that a parameter proposition will not be fulfilled in the future are examined in Section 4.5.1.4.

[3]There is one exception: If the repeated time point is a parameter change and we know for some reason, that this parameter will never change again, then we know that there will not be another instance of the time annotation with a new reference point. However, determining this case is a complex task and the consequences are negligible in practice.

4.5.1.3 Temporal pattern

Temporal patterns are the heart of the conditions governing the change of plan states. Besides other things not related to data abstraction, temporal patterns can contain the following.

- A single parameter proposition.
- A plan state constraint.
- Several temporal patterns connected by Boolean operators.
- Two temporal patterns connected by qualitative temporal operators such as *before* or *during*. Asbru implements 7 of the 13 temporal relations defined by Allen [8]. The missing 6 are inversions of other relations which can easily be achieved in Asbru by swapping the arguments.
- A count constraint consisting of a temporal pattern and a number specifying how often the temporal pattern must be fulfilled to fulfil the count constraint.

Monitoring a count constraint is implemented by a simple comparison of the number of episodes as describe in Section 4.7. Monitoring the three varieties of parameter propositions is described in Section 4.5.2.1 through 4.5.2.3. Monitoring temporal relations is described in Section 4.5.4.

4.5.1.4 Plan and condition

The treatment is formalized as a hierarchy of skeletal plans in Asbru. These plans are stepwise refine by selecting appropriate children for each plan at runtime.

A plan in Asgaard has various states. The change between states is governed by conditions. These conditions are built from temporal patterns (and other things which have no connection to data abstraction).

In managing plan states, the fact that a condition will never be fulfilled has consequences. Table 4.6 gives a short overview of the conditions and the consequences of fulfilling or not fulfilling them.

To distinguish the success or failure of a plan from that of a single condition, a condition *succeeds* if it is fulfilled and it *fails* if it cannot be fulfilled in the future and never was in the past. In contrast, a plan *aborts* if its abort condition succeeds and *completes* if its complete condition succeeds.

Conditions contain parameter proposition which contain time annotations. The shifts in these time annotations can contain references to parameters, variables, and arguments in addition to constants. Therefore, their value is not known at program start, but only when the plan containing them comes into life.

To cope with this situation, the ParameterPropositionModules described in the following have a method *startMonitoring* which is called with the content of the time annotation as soon as the plan is instantiated[4] and these values are known. Until then, the inputs to the ParameterPropositionModule are stored in an intermediate buffer.

[4]Note that on the implementation level, the PlanStateModule is created at program start, while on the concept level, the *plan* is instantiated only when its parent is a the right state to do so.

condition	plan state if condition is		
	fulfilled	not yet fulfilled	never more fulfilled
filter	possible	considered	rejected[1]
setup	activated	possible	rejected[2]
suspend	suspended	activated	no impact
reactivate	activated	suspended	no immediate change[3]
complete	completed	activated	no immediate change[3]
abort	aborted	activated or suspended	no practical impact[4]

[1] Frequent and important. The filter condition is often used to choose from a set of alternatives and it is crucial for the functioning of the plan execution to know which plans can be ruled out.

[2] This case is very rare. It happens if the plan execution unit can rule out any action it could take to fulfil the setup condition.

[3] If a plan cannot compete in the future, then the plan enters a zombie state in practice. However, its formal state is not changed since this would overly complicate the semantics of plan states. If the plan designer wants the plan to abort under this condition, the abort condition (and the time annotation of the plan activation) can easily be adjusted accordingly.

[4] A plan which will never abort in the future cannot be considered to be completed. Therefore this bit of positive information is not used in practice.

Table 4.6: Condition and the consequences of their being fulfilled or not.

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 4.7: Elements of a time annotation and their abbreviations.

4.5.2 Monitoring parameter propositions

In this section, we focus on monitoring the temporal aspect of the parameter proposition. The part which maps the value description is implemented by different modules and outputs a Boolean which we call *the condition* here. E.g., if the parameter proposition specifies "A greater 10 for a minimum duration of 1 hour", then comparing A against a constant value of 10 is performed outside the scope of this subsection and it will yield a stream of Booleans which are true in all time steps at which A is greater 10. In this and the following subsections we focus on the minimum duration of 1 hour, and efficient and delay-free algorithms to detect the fact that we found an interval at which the condition is fulfilled and which fulfils the temporal constraints set forth in the time annotation.

The time annotation contained in a parameter proposition can have three types of reference point:

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

The principal design is a finite state machine in each of the cases, which reacts on events caused by both changes in the input and elapse of an interval of time. However, the algorithms differ significantly depending on the type of reference point. Therefore, they are described in three distinct subsections, starting with the simplest case, the fixed reference point, continuing with the second case above, which builds on the first, and finishing with the moving time point now.

4.5.2.1 Monitoring parameter propositions with a fixed reference point

First, we concentrate on a time annotation which specifies the start and the end of the interval to find. It is not necessary that the time annotation is complete, i.e., that all six shifts are given, but there must be enough information to determine start and end of the interval constraining the start and end of the searched interval. Details on the complex dependencies of the six shifts are explained below.

Table 4.7 repeats the elements of the time annotation and their abbreviations for convenience.

In addition, the following abbreviations are used in the following:

PF: The first time point at which *the condition* is fulfilled.

NF: The first time point at which it is no more fulfilled.

For a fixed reference point RP, the following definitions are useful:

Earliest starting time (EST): The absolute time point computed by adding ESS and RP, or negative infinity if ESS is not specified.

Latest starting time (LST): The absolute time point computed by adding ESS and RP, or positive infinity if LSS is not specified.

Earliest finishing time (EFT): The absolute time point computed by adding ESS and RP, or negative infinity if EFS is not specified.

Latest finishing time (LFT): The absolute time point computed by adding ESS and RP, or positive infinity if LFS is not specified.

In the monitoring process we observe four different and independent processes:

1. The passing of the interval for matching positive flanks ([EST, LST]).
2. The passing of the interval for matching negative flanks ([EFT, LFT]).
3. The passing of one interval per positive flank, based on the duration. Each interval ([PF + MinDu, PF + MaxDu]) constrains the corresponding negative flank in addition to the above.
4. The state of *the condition*, i.e., the sequence of positive and negative flanks as they result from matching the value description to the actual values of the parameter over time.

Note that if the first two intervals are sufficiently long, several positive flanks can occur, creating several instances of [PF + MinDu, PF + MaxDu].

Not all time annotations need to constrain both sides of both the interval for the starting time and that for the finishing time. Undefined values can lead to EST and/or EFT being infinitely early, PF + MinDu coinciding with PF, and LST and/or LFT and/or PF + MaxDu being infinitely late. For the monitoring process, this has the following consequences.

EST: No PF is required for the first interval – the value description simply must be fulfilled at the start of monitoring. This is implemented by adding a dummy PF at the start of monitoring (but after the infinitely early EST) if the parameter meets the value description at the very first measurement, i.e., if a potentially fitting interval started before program start[5]. In addition, the monitoring process must allow the transition resulting from EST to occur indefinitely early, i.e., before the processing of the dummy PF.

LST: Any further PF will be accepted, but LFT may put an end to monitoring.

[5]Note that this only refers to actual start of the data abstraction software, not the start of a plan.

EFT: Any NF (before LFT) is suitable. As for EST, the monitoring process must implement the transition of EFT before processing any flanks.

LFT: The latest time for NF is only constrained by MaxDu + PF. Since this constraint is renewed at each PF, there is no absolute constraint for the latest NF. However, LST may constrain the latest PF, imposing an absolute end of monitoring.

MinDu: The minimum duration is EFT - PF, or 0 if EFT already passed.

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

LST and LFT: There is always a chance that a matching interval can be found in the future and thus waiting for it never ends.

LFT and MaxDu: Once a suitable PF is found, the interval is a fitting interval already at EFT (if no NF occurred between PF and EFT) because already then all requirements are fulfilled (NF cannot come too late).

If both EFT and LFT are undefined, the result does not differ from the sum of the effects described for each of them. The same holds for EST and EFT. I.e., these cases do not call for special rules.

Figure 4.12 shows a state chart for monitoring parameter propositions with fixed reference point. As mentioned above, four intervals resp. states are monitored in parallel:

1. The starting interval SI during which PF must occur to start a fitting interval. It is started by the event of EST arriving and ended by LST. The state of being in this interval is represented by the dotted rectangle at the left labelled “SI = during”. LST causes a shift from the left half of the diagram (states 2, 3, 7, and 8) to the right half of the diagram (states 4, 5, 9, and 10) which mirror the first group, with the difference that after the end of the current interval the monitoring process terminates while on the left half of the diagram it continues.

2. The finishing interval FI during which NF must occur to end a fitting interval. It is started by EFT and ended by LFT. The state of being in this interval is represented by the dotted rectangle at the lower half of Figure 4.12 labelled “FI = during”. LFT leads to the ultimate termination in any state.

3. The interval defined by MinDu and MaxDu, DI. It is started by the event of MinDu being over and ended by either LFT, MaxDu, or NF. It is represented by a U-shaped polygon labelled “DI = during”.

4. The state of having found a fitting PF, i.e., during a potentially fitting interval. It is started by PF and ended by many different events, among only NF during both FI and DI leads to a positive result. It is represented by a dotted rectangle in the centre of the diagram, labelled “PFfound = true”.

Implementing figure 4.12 node by node and arc by arc as a finite state machine seems somewhat redundant. Figure 4.13 shows pseudo code for an implementation which stores the state information in several independent variables described in the caption on this figure.

The algorithm produces four types of messages or output records:

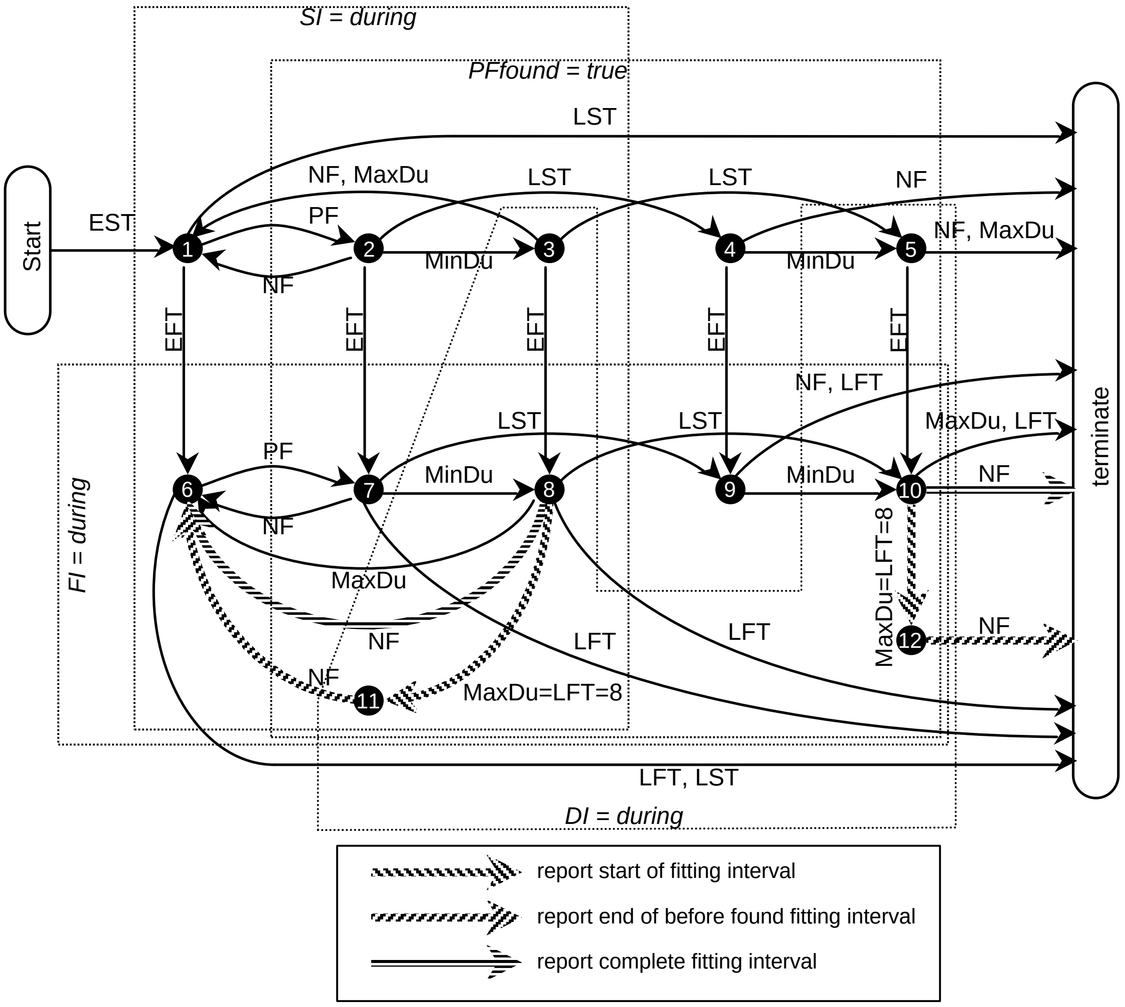


Figure 4.12: Full state chart of monitoring a parameter proposition with fixed reference point. Wide, stripped arcs stand for the action "output description of fitting interval" performed during these state transitions. This action is split in cases where the interval is found to fit before it ends. Compare description on page 100.

```
start:     if EST = $-\infty$
           then SI := during
           else  SI := before
           if EFT = $-\infty$
           then FI := during
           else  FI := before
           PFfound := false
EST:       SI := during
LST:       if PFfound
           then SI := past
           else  *terminate*
EFT:       FI := during
LFT:       *terminate*
PF:        if SI = during
           then PFfound := true
                *set events* MinDuEx *and* MaxDuEx
                DI := before
MinDuEx:   DI := during
           if FI = during and MaxDu = $\infty$ and LFT = $\infty$
           then *report start of interval*
NF:        if PFfound and FI = during and DI = during
           then if MaxDu = $\infty$ and LFT = $\infty$
                then *report end of before found interval*
                else  *report complete interval*
           if SI = during
           then PFfound := false
           else  *terminate*
MaxDuEx:   DI := past
           if SI = during
           then PFfound := false
           else  *terminate*
```

Figure 4.13: Pseudo code for monitoring parameter propositions based on a fixed reference point.
State variables: SI = Starting Interval [EST, LST], FI = status of Finishing Interval [EFT, LFT], DI = status of interval of duration constraints [PF + MinDu, PF + MaxDu], PFfound = most recent PF was in starting interval.
Events: EST = Earliest Starting Time, LST = Latest Starting Time, EFT = Earliest Finishing Time, LFT = Latest Finishing Time, PF = Positive Flank, NF = Negative Flank, MinDuEx = Minimum Duration Expired, MaxDuEx = Maximum Duration Expired, start = initialization done before processing first input.
Note: "MaxDu = ∞ and LFT = ∞" as well as MinDuEx are checked immediately after other state transitions, i.e., more then one transition can occur at a given value of valid time.

- An interval just ended and now is found to fit. Therefore both PF and NF are known (and output).
- An interval is found to fit, but it has not ended yet. Therefore only PF is known. Still it is important to report it immediately, since the plan execution unit might wait for it and reacting to this interval at its end may be far too late.
- The current fitting interval which has been reported before just ended. In addition to the before output PF, now NF is known too.
- End of monitoring has arrived, either because of LST (and no interval currently open) or LFT. This information may be used by the plan execution unit to deduce that a certain condition will never be fulfilled.

4.5.2.2 Monitoring parameter propositions with repeated reference points

Monitoring such parameter propositions consists of monitoring a parameter proposition with fixed reference point several times, each time with a different reference point. The union of all episodes found in all of the monitoring sub-processes is the result of monitoring the parameter proposition with repeated reference point.

On the implementation level, this task consists of

- instantiating the repeated time points,
- starting a monitoring process for each, and
- passing the union of all episodes found by each of the sub-processes on to the output.

Truly repeated time points are defined by *state-change* elements, which describe a qualitative parameter, a qualitative value for it, and whether leaving or entering this state should produce a repeated time point. On the implementation level, these time points are produced by a StateChangeModule. The output of each of them is feed into the RepeatedParameterPropositionModule.

There is also the case that several reference points have been specified in the time annotations. Their count is fixed and for each of them a ParameterPropositionModule is instantiated as above and their output is merged. The main difference is that monitoring stops after monitoring the last of these parameter propositions stopped. This is in contrast to monitoring parameter proposition with a truly repeated reference point, where monitoring does not stop.

"Merging" the episodes does not mean that intervals are joined (in most cases they could not because of gaps between them), it means that duplicates are removed from the union of all episodes delivered by the monitoring sub-processes. Consider the example "X is high for at least 5 minutes during the hour after Y became low". The Asbru code is shown in Figure 4.14, while Figure 4.15 shows an example of episodes where the middle one is a fitting interval for both instances of this repeated parameter proposition.

The implementation of finding duplicates is simple, since duplicates must have the same PF.

```
parameter-proposition parameter-name=”X”
    value-description type=”equal”
        qualitative-constant value=”high”
    context
        any
    time-annotation
        time-range
            starting-shift
                earliest
                    numerical-constant value=”0”
            finishing-shift
                latest
                    numerical-constant value=”60” unit=”min”
            duration
                minimum
                    numerical-constant value=”5” unit=”min”
        parameter-change value=”low” direction=”enter”
            parameter-ref name=”Y”
```

Figure 4.14: A time annotation with repeated time point, matching all episodes of X being *high* for at least 5 minutes starting not before Y became *low* and finishing 60 after Y became *low* at the latest.

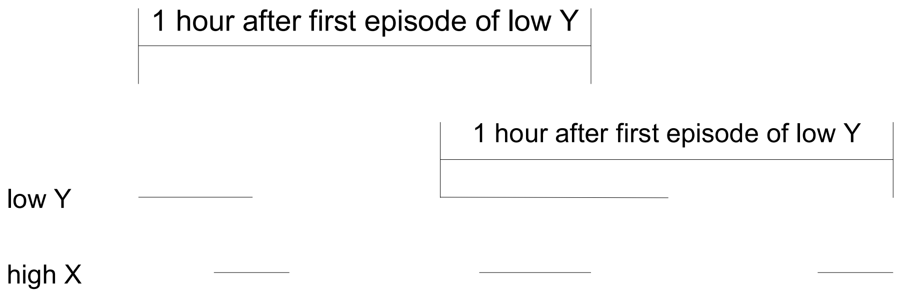

1 hour after first episode of low Y
1 hour after first episode of low Y
low Y
high X

Figure 4.15: Episodes of X being *high* within 1 hour after Y became *low*. Note that the middle episode of X matches both parameter proposition, the one based on the first reference point and the one based on the second reference point, since it falls within both 1-hour-intervals.

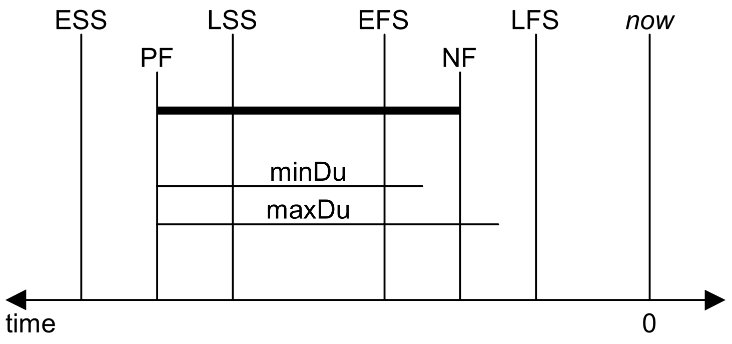


Figure 4.16: A parameter proposition based on now and a interval fitting at the moment shown.

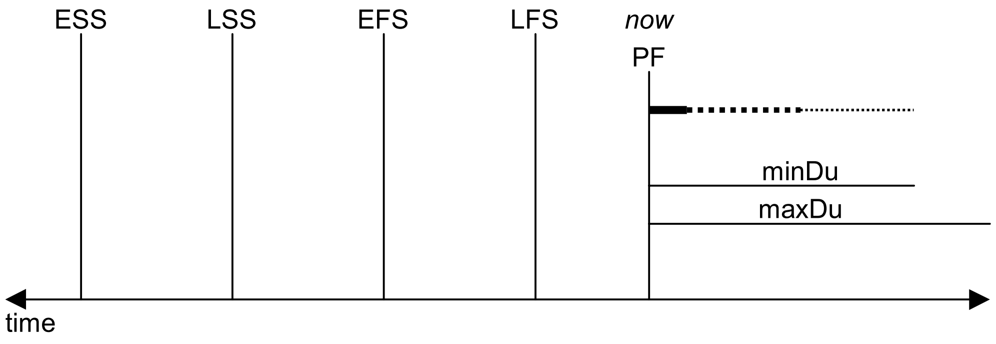


Figure 4.17: A potentially fitting interval at the moment now=PF.

4.5.2.3 Monitoring parameter propositions with reference point now

The moving time point now calls for a completely different approach then the above cases. Now, we can no longer calculate EFT etc. in advance. Also, there is no end of monitoring. And finally, shifts must be negative or zero, i.e., lying in the past or in the presence. Figure 4.16 shows a prototypical case of a time annotation based on now and a fitting interval. Note that the fitting interval moves to the left with respect to now.

For the above mentioned reasons, the monitoring algorithm focuses on the flanks. Figures 4.17 to 4.20 illustrate the important phases in comparing an episode to a fully specified time annotation based on now. At the positive flank (PF) we calculate the time points between which the negative flank (NF) should lie to fulfil the constraints set forth by MinDu and MaxDu (Figure 4.17). If NF occurs within these constraints (Figure 4.18), we know that they are satisfied, but the parameter proposition is not fulfilled *yet* – both PF and NF lie *past* the intervals constraining them.

As time goes by, i.e., as the value of now increases, the time points reflecting these shifts move to the right relative to the episode. After some time, both PF and NF enter the intervals constraining them (Figure 4.19). In this figure, both flanks enter their constraining intervals simultaneously, but this need not be the case. The first value of now, for which all constraints are satisfied, is called the *start of validity (SV)*. Its

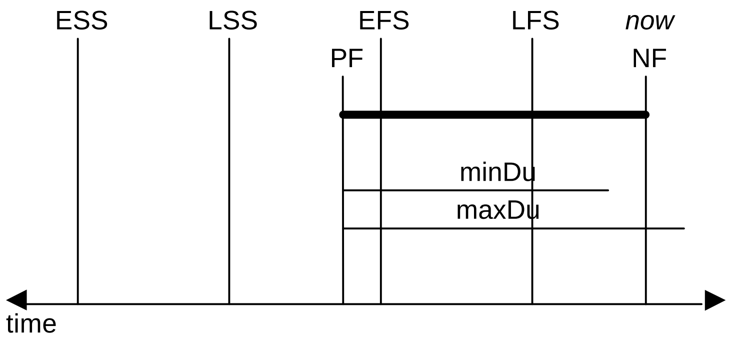


Figure 4.18: A future fitting interval at the moment now=NF.

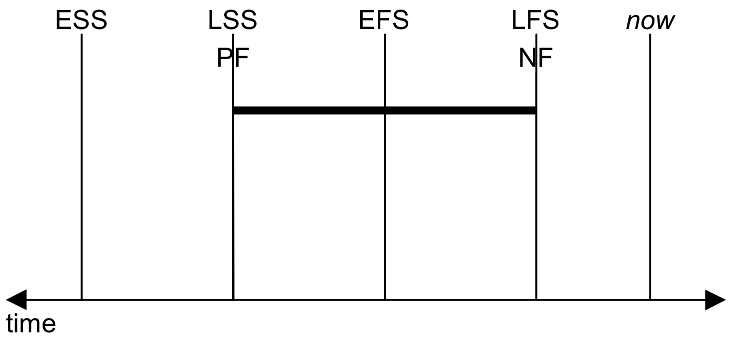


Figure 4.19: A fitting interval at the first (lowest) value of now for which it fits – at *Start of Validity (SV)*.

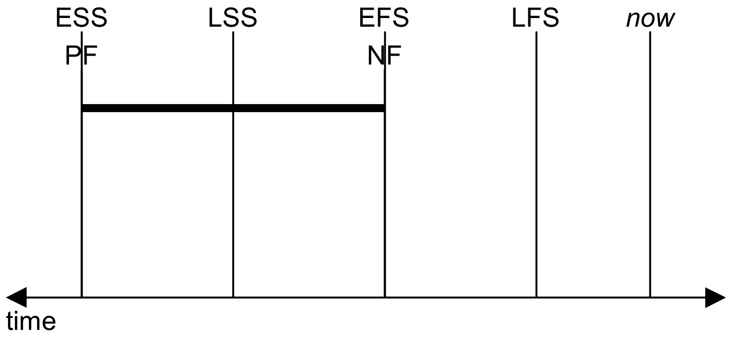


Figure 4.20: A fitting interval at the last (highest) value of now for which it fits – at *End of Validity (EV)*.

formal calculation is given below.

As now increases further, the first of both flanks will leave the constraining interval (Figure 4.20). Again, this happens simultaneously in the example which need not be the case for other time annotations. The value of now at this moment is the *end of validity (EV)*. For greater values of now, the parameter proposition is *no more* fulfilled.

Calculating start and end of validity. Both SV and EV can be derived from the standard constraints for parameter proposition. Replacing the reference point by now, we arrive at the following constraints for the flanks.

$$\begin{array}{rcccl} now + ESS & \leq & PF & \leq & now + LSS \\ now + EFS & \leq & NF & \leq & now + LFS \\ MinDu & \leq & NF - PF & \leq & MaxDu \end{array}$$

Any lower limit of now derived from these constraints constitutes a constraint on the start of validity, any upper limit constrains the end of validity. ESS, EFS, and MinDu together with the flanks form the maximum for now, i.e., EV, while LSS and LFS determine SV.

MinDu influences SV independent from now. Since we cannot state that $NF - PF > MinDu$ before PF + MinDu, we cannot know that the interval fits earlier. MaxDu does not directly influence the duration of the validity interval – if it is given, NF must be awaited and if NF arrives past PF + MaxDu then the interval is not fitting at all, i.e., for no value of now.

$$\begin{array}{rcl} EV_{\mathrm{PF}} & = & PF - ESS \\ EV_{\mathrm{NF}} & = & NF - EFS \end{array} \qquad \begin{array}{rcl} SV_{\mathrm{PF}} & = & PF - LSS \\ SV_{\mathrm{NF}} & = & NF - LFS \\ SV_{\mathrm{MinDu}} & = & PF + MinDu \end{array}$$

Since all five constraints must be observed, the valid interval is defined by SV and EV as:

$$\begin{array}{rcl} SV & = & \max(SV_{\mathrm{PF}}, SV_{\mathrm{NF}}, SV_{\mathrm{PF}}) \\ EV & = & \min(EV_{\mathrm{PF}}, EV_{\mathrm{NF}}) \end{array}$$

For the monitoring process, the question whether EV can lie before SV is important. In such a case, we must not report a fitting interval at SV, which for other cases is very desirable. $EV < SV$ if any of the terms constituting SV is greater then any of the terms constituting EV. We therefore must examine 6 cases.

1. SV$_{\mathrm{PF}}$ is never greater EV$_{\mathrm{PF}}$:

$$\begin{array}{rcll} ESS & \leq & LSS & \rightarrow \\ PF - ESS & \geq & PF - LSS & \rightarrow \\ EV_{\mathrm{PF}} & \geq & SV_{\mathrm{PF}} & \end{array}$$

2. SV$_{\mathrm{NF}}$ is greater EV$_{\mathrm{PF}}$ if the duration of the interval exceeds LFS - ESS. As discussed in Section 4.5.1.2, this value is a hidden maximum duration constraint.

While it is handled implicitly by the algorithm for detecting parameter propositions with fixed reference point, here it needs special attention (see below).

$$\begin{array}{rcll} SV_{\text{NF}} & > & EV_{\text{PF}} & \leftrightarrow \\ NF - LFS & > & PF - ESS & \leftrightarrow \\ NF - PF & > & LFS - ESS & \end{array}$$

3. SV$_{\text{MinDu}}$ is greater then EV$_{\text{PF}}$ if MinDu is greater the absolute value of ESS. This is a form of malformation of the time annotation (specific to time annotations with reference point now). It means that, PF must not lie further in the past then ESS and at the same time we must wait for a greater value MinDu after PF to be sure that NF does not occur before PF + MinDu.

$$\begin{array}{rcll} SV_{\text{MinDu}} & > & EV_{\text{PF}} & \leftrightarrow \\ PF + MinDu & > & PF - ESS & \leftrightarrow \\ MinDu & > & -ESS & \end{array}$$

4. SV$_{\text{PF}}$ is greater then EV$_{\text{NF}}$ if the duration of the interval is smaller than EFS - LSS. This means that – as for parameter propositions with fixed reference point – EFS - LSS imposes a lower limit for MinDu. An other words, this case does not happen for well-formed time annotations, since $MinDu \geq EFS - LSS$ is one of the criterions for being well-formed.

$$\begin{array}{rcll} SV_{\text{PF}} & > & EV_{\text{NF}} & \leftrightarrow \\ PF - LSS & > & NF - EFS & \leftrightarrow \\ EFS - LSS & > & NF - PF & \end{array}$$

5. SV$_{\text{NF}}$ is never greater EV$_{\text{NF}}$:

$$\begin{array}{rcll} EFS & \leq & LFS & \rightarrow \\ NF - EFS & \geq & NF - LFS & \rightarrow \\ EV_{\text{NF}} & \geq & SV_{\text{NF}} & \end{array}$$

6. SV$_{\text{MinDu}}$ is greater EV$_{\text{NF}}$ if the interval is shorter than MinDu under the (proper) assumption that EFS is not greater zero in time annotations based on now.

$$\begin{array}{rcll} SV_{\text{MinDu}} & > & EV_{\text{NF}} & \leftrightarrow \\ PF + MinDu & > & NF - EFS & \leftrightarrow \\ MinDu + EFS & > & NF - PF & \xrightarrow{EFS \leq 0} \\ MinDu & > & NF - PF & \end{array}$$

This leads to the following conclusions. First, there is an additional constraint for a time annotation based on now to be well-defined, which do not exist in time annotations based on a fixed reference point: $-ESS \geq MinDu$. It this is violated, then the validity interval of the time annotation ends before MinDu is over. Therefore, the monitoring process must fail.

The second conclusion lies in the definition of EffMaxDu and EffMinDu as follows. The term 0 is included for clarity only, since 0 is the default value for undefined MinDu.

$$\begin{aligned} EffMaxDu &= \min(LFS - ESS, MaxDu) \\ EffMinDu &= \max(EFS - LSS, MinDu, 0) \end{aligned}$$

While $LFS - ESS \geq MaxDu$ is a trivial constraint for the wellformedness of any time annotation, the above formula is important in those cases, in which one or more of the three values are undefined. MaxDu and EffMaxDu play an important role in the monitoring process only through their potential absence. If there is no limit for the latest occurrence of NF, then the fitting interval must be reported before it is completed. As for parameter propositions with fixed reference point this is the case, if both MaxDu and LFS are undefined.

A two-dimensional view. Figures 4.17 to 4.20 seem intuitive, but space consuming. Figure 4.21 integrates both the movement of the flanks and 4.20 to now and the calculation of the validity interval into one two-dimensional diagram. It shows three different dimensions on three temporal axes:

1. From the origin to the right the absolute time at which flanks occur is plotted. This is the valid time of the data points representing them.
2. From the origin to the left, the shifts are plotted. Values on this side of the axis are relative to now. All shifts in a time annotation based on now are negative (or zero).
3. On the vertical axis the time of monitoring is shown. This is equal to the values which now takes during the monitoring process.

The diagonal lines originating from the shifts and ascending at a slope of 1:1 show how the absolute time points associated with them change over time. With each step in time, they are shifted to the right relative to the absolute time axis on which the flanks are plotted. This is the opposite point of view compared to Figures 4.17 to 4.20 where the shifts are the reference and the flanks move to the left. Still, the depicted process is the same.

PF is constrained by ESS and LSS. Hence, the vertical position of the intersections between the vertical line representing PF and the diagonals representing ESS and LSS mark the end and start of validity defined by this flank (EV$_{PF}$ and SV$_{PF}$) it projected to the vertical axis. The same holds for NF. The actual interval of validity is constrained by the later of both SV and the earlier of both EV.

The constraints EffMinDu and EffMaxDu are plotted on the horizontal axis since they hold positive values in contrast to the shifts. EffMinDu is represented by a diagonal ascending at 1:1 which intersects the vertical axis MinDu units above the origin. A horizontal line plotted through the intersection of this diagonal with the vertical line at PF shows SV$_{\mathrm{MinDu}}$.

EffMaxDu does not constrain the interval of validity.

The projection of both EffMinDu and EffMaxDu on the axis of the flanks is show in dotted lines. Drawing a diagonals descending at a rate of 1:1 from the intersections

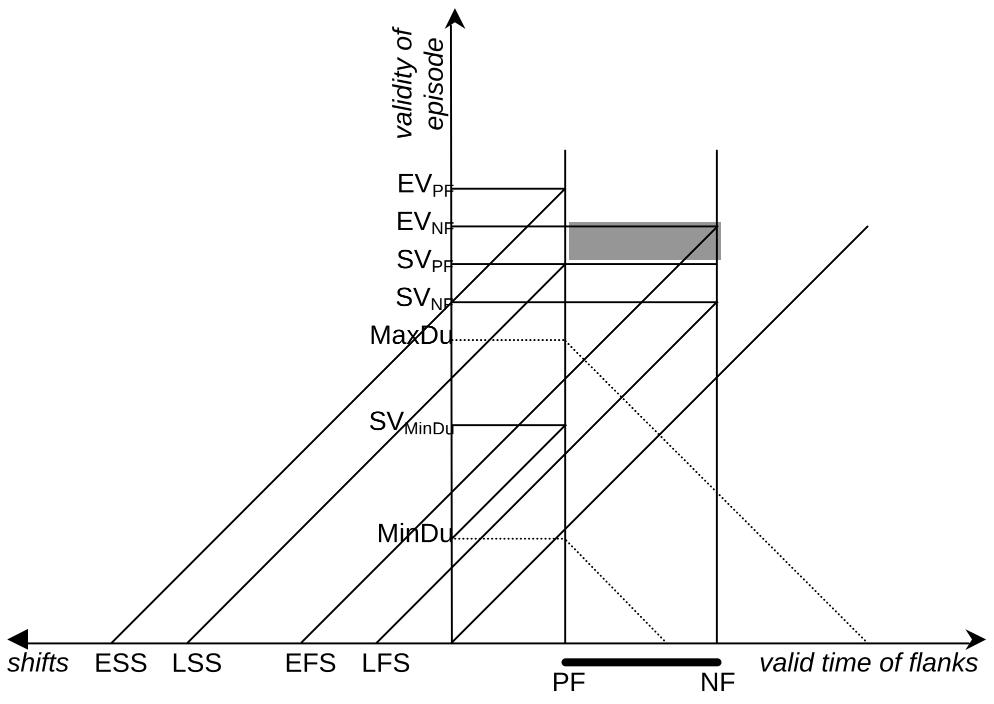


Figure 4.21: Two-dimensional view of matching an interval to a time annotation based on now.
Usage of the axes: To the right: absolute time at which flanks occur; to the left: the shifts given in the time annotation; up: the time of monitoring, i.e., the values of now as monitoring progresses. The height of the grey rectangle shows the time during which the parameter proposition is considered fulfilled.
EV$_{PF}$ and EV$_{NF}$: End of Validity based on Positive and Negative Flank.
SV$_{PF}$ and SV$_{NF}$: Start of Validity based on Positive and Negative Flank.
SV$_{\mathrm{MinDu}}$: Start of Validity based on Minimum Duration.
PF and NF: Positive and Negative Flank for one instance of input interval.

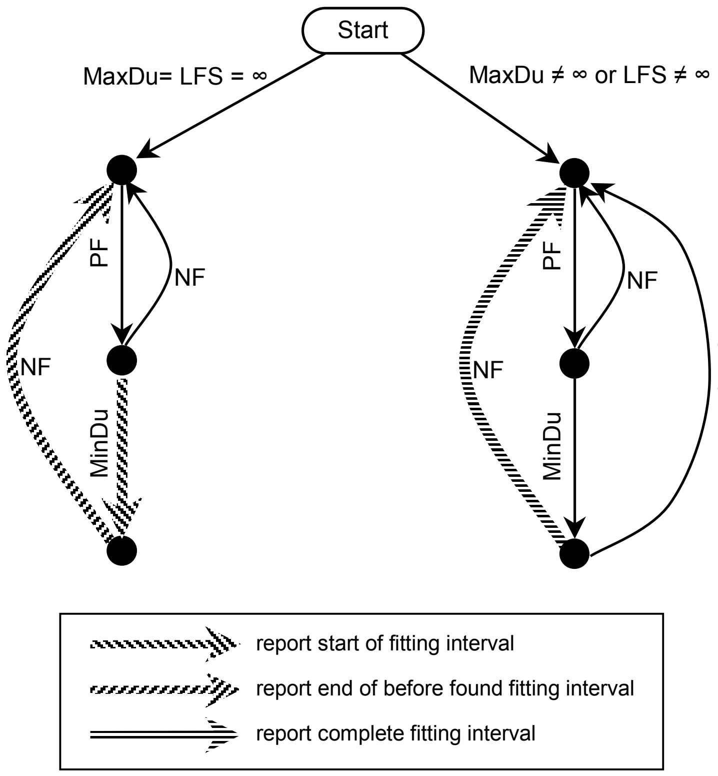

Start
MaxDu= LFS = ∞
MaxDu ≠ ∞ or LFS ≠ ∞
PF
NF
NF
MinDu
PF
NF
NF
MinDu
MaxDu
report start of fitting interval
report end of before found fitting interval
report complete fitting interval

Figure 4.22: State chart for monitoring parameter propositions based on now.
Events: PF = Positive Flank, NF = Negative Flank, MinDu = PF + EffMinDu arrived, MaxDu = EffMaxDu arrived.

of the vertical line representing PF and horizontals representing EffMinDu and EffMaxDu shows these constraints of NF on the horizontal axis. These considerations do not contribute to the calculation of SV and EV, but rather to the selection of NF as a proper end of a fitting interval.

A monitoring algorithm. There are two principal modes of the monitoring algorithm, depending on whether both MaxDu and LSS are undefined, i.e. positive infinite, or not. In the first case, SV can arrive before NF and at SV at the latest the incomplete fitting interval must be reported. More precisely, after PF we must await PF + Mindu. If NF occurs before, then this interval was too short. Otherwise, we found a fitting interval with $SV = \max(PF - LSS, PF + MinDu)$ which will be now if $-LSS < MinDu$ or in the future (at $PF - LSS$).

In the second case, if MaxDu is given, NF must be awaited to see whether it occurs before PF + MaxDu. Otherwise the interval would not fit. If LFS is given, SV cannot lie before NF; And NF must be awaited to calculate SV$_{\mathrm{NF}}$.

$$\left.\begin{array}{lcl} SV = \max(\ldots, SV_{\mathrm{NF}}, \ldots) & \rightarrow & SV \geq SV_{\mathrm{NF}} \\ SV_{\mathrm{NF}} = NF - LFS & \rightarrow & SV_{\mathrm{NF}} \geq NF \end{array}\right\} \rightarrow SV \geq NF$$

Figure 4.22 shows the algorithm for monitoring parameter propositions with reference point now in the form of a state chart. Note that – as for the state charts above –

```
Start:   PFfound := false
         if     MaxDu = ∞ and LSS = ∞
         then   noMaximum = true
         else   noMaximum = false

PF:      PFfound := true
         MinDuPassed := false
         set MinDu and MaxDu events

NF:      PFfound := false
         if     MinDuPassed
         then   if     noMaximum
                then   report end of before found interval
                else   report complete interval

MinDu:   MinDuPassed := true
         if     noMaximum
         then   report start of interval

MaxDu:   PFfound := false
```

Table 4.8: Pseudo code for monitoring parameter propositions based on *now*.
Events: PF = Positive Flank, NF = Negative Flank, MinDu = PF + EffMinDu arrived, MaxDu = EffMaxDu arrived.

the automaton must be able to take several steps in immediate succession. Also, some arcs are labelled with queries while most are labelled with events. Therefore, the figure is an illustration, not a formal specification.

Table 4.8 shows the algorithm as pseudo code.

4.5.2.4 Integrating the output of different types of parameter propositions

Above, three dissimilar monitoring processes were described. In order to process their output in a satisfying way, it is necessary that this output is as uniform as possible, independent of the type of reference point in their time annotation. While Section 4.1 describes the integration of all modules, the specific integration of parameter proposition modules is discussed in the following, since it forms the basis of monitoring combinations of temporal patterns as discussed in Sections 4.5.4 and 4.5.5.

Temporal dimensions of monitoring results. There are two independent temporal intervals:

1. The temporal episode, i.e., the interval starting at PF and ending at NF.
2. The time, during which this episode is considered to be a fitting interval.
   For fixed reference points (including repeated ones) this starts with the moment at which the fact, that this interval fits is known. There is no end to this interval for parameter propositions with fixed reference points.
   For the moving reference point *now*, the interval spans from SV to EV, and both

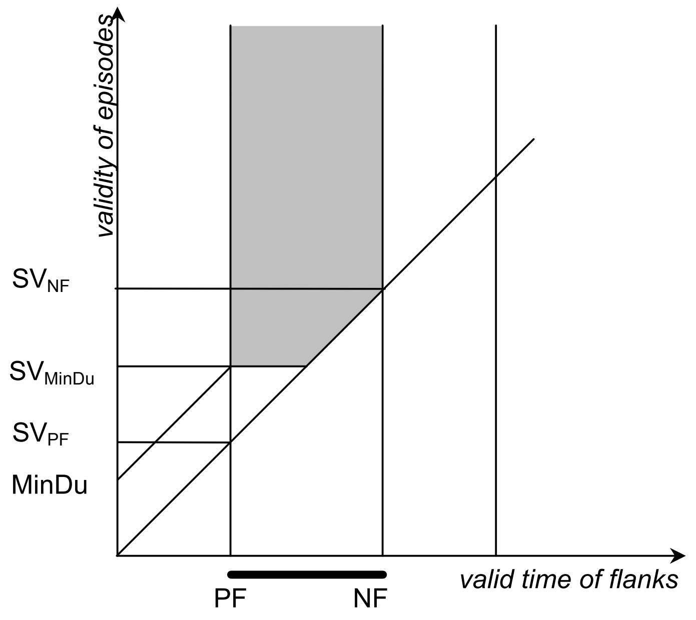


Figure 4.23: Extent of the episode (horizontally) and time of its validity (vertically) for a parameter propositions with fixed reference point.
PF = Positive Flank, NF = Negative Flank, MinDu = Minimum Duration defined in Time Annotation.
SV$_{PF}$ and SV$_{NF}$: Start of Validity based on Positive and Negative Flank.
SV$_{\mathrm{MinDu}}$: Start of Validity based on Minimum Duration.
SV$_{NF}$ is only relevant if the end of the episode is constrained.
SV$_{\mathrm{MinDu}}$ is only relevant if MinDu is given.
The effective Start of Validity is the maximum of all relevant SVs.
Once established, the validity of an episode matching a parameter proposition with fixed reference point never terminates.

are defined by the shifts of the time annotation. SV often lies in the future at the moment at which the interval is found fit, and it can never lie in the past per definition.

The second interval can be split into a first part, during which only PF is known (but the interval is known to fit) and a second part in which both PF and NF are known.

Each parameter proposition can match more than one episode.

Figure 4.23 shows the relation between valid time of the measurements which specifies PF and NF on the one hand, and the time during which certain episodes are considered to be matching on the other hand, for a time annotation with fixed reference points. The cut off corner of the rectangles reflects the time at which the end of the interval is not known.

Figure 4.24 shows the same relation for a similar time annotation based on now. Note that the projections need to be different since now is moving, but the basic shape is the same. However, due to the possible short duration of the validity of the interval (i.e., the extent along the vertical axis) here we can see cases like the depicted one, where the “rectangle” does not extend to the full width of NF - PF. Instead it is cut off

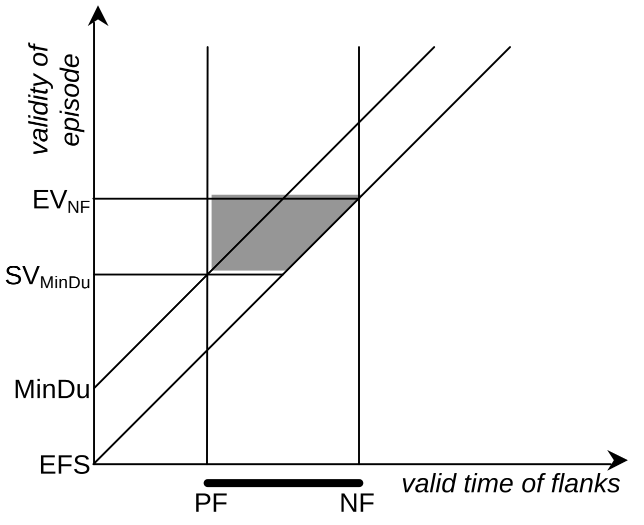


Figure 4.24: Extent of the episode (horizontally at the right) and time of its validity (vertically) for a parameter propositions based on now.

before by the diagonal line representing now = *valid time*.

Usage of monitoring results. The following questions summarize the capabilities of the feature abstraction modules deployed on the found episodes, i.e., the information for which the monitoring process must provide the basis.

- Was there a fitting interval?
- How many fitting intervals are there?
- What is the duration of each of them?
- When does a fitting interval start, when does it end?

Was there a fitting interval? This is easily answered by producing a *true* value at SV for the first (or every) episode which fits. At the end of monitoring, a *false* value is produced. At the start of monitoring, an *undefined* value is produced. This is easily implemented using Asbru Boolean values – they can be *undefined*.

The recipients of this information are abstraction modules such as Boolean combinations (described in Section 4.5.5) on the one hand and the plan execution unit which changes a plan state accordingly and ignores any further details on the other hand. The plan execution unit ignores EV when it arrives, since the change in plan state cannot be made undone[6], but the Boolean combinations do not.

Note that for the behaviour of the plan execution unit the valid time of the data points received is not relevant – it reacts at transaction time[7].

[6]If desired, this can be modelled explicitly by stating it in another condition.

[7]If the valid time lies in the future since SV is known in advance, no action is taken until valid time equals now.

How many fitting intervals are there? The solution is clearly to increment the count at each SV and decrement it at each EV. But there is an important side question: When were there how many? Suppose we ask how many episodes of high fever for at least 1 hour were there? If there is an episode of 2 hours of high fever. Does the count of such episodes change at PF or at SV?

The solution is to set the transaction time to SV and EV and the valid time of the change of count to the start and end of the interval, i.e., the valid times of PF and NF. Such a solution satisfies both plan execution unit and further abstraction modules analyzing the past as precisely as possible.

What is the duration of each of them? Here, the duration is clearly measured from PF to NF. But it is only known after NF. So for intervals with unconstrained end (more precisely for time annotation with undefined LFS and MaxDu) the duration is increasing as time passes while waiting for NF. This is represented by the diagonal border at the lower right of the filled polygons in Figures 4.23 and 4.24. To be practical, the module monitoring the parameter proposition should produce output when something changes in a qualitative way, e.g., if NF occurs but not each second to announce that the duration until now increased by one second. This can be performed much better by other abstraction modules extracting the features from the episodes.

Since episodes matched to a parameter proposition based on now lose their status as fitting interval at EV, abstraction modules working with the duration must be informed then, since the duration of the episode in question might be considered zero after EV.

When does a fitting interval start? When does it end? Here it is clear that PF and NF are accessed and not SV and EV.

Summary. The demand can be satisfied by the following unified output.

- Each episode is described by one or more objects (of class EpisodeDataPoint) of the same format with equal episode identifier. The fields of this object are PF, NF, and episode identifier (ID). If at some time NF is not known, it is set to the *undefined* value.
- At SV a first object (with new ID) is output, possibly with undefined NF.
- At NF another object with same ID is output, this time with NF set.
- At EV an object with the same ID and both PF and NF undefined is output.
- At the end of monitoring an object with undefined ID is output to show this special (terminal) situation.

As shown in the next subsections, this format is not only suitable to describe episodes found by parameter propositions but also such found be temporal patterns.

A before B	A ends before B starts
A overlaps B	A ends after B starts
A starts B	A and B start at the same time
A equals B	Both start and end of A and B are exactly the same
A meets B	A ends at the same time B starts
A during B	A starts after B starts and ends before B ends
A finishes B	A and B end at the same time

Table 4.9: The set of Allen's temporal relations implemented in Asbru.

4.5.3 Monitoring plan state constraints

Plan state constraints contain a plan state, a plan pointer, and a time annotation. They are fulfilled, if the plan referred to was in the plan state given during the interval defined by the time annotation.

The only difference to a parameter proposition applied to a qualitative parameter is the source of the monitored values. They are supplied by a PlanStateExtractionModule, which transforms the complex plan-state data-point output by PlanStateModules into a qualitative value.

Otherwise, the monitoring process is exactly the same as for parameter propositions, including the implications of different types of reference points.

4.5.4 Monitoring temporal constraints

A temporal constraint consists of two temporal patterns and a temporal relation between them. It is fulfilled if at least one combination of fitting intervals of the two temporal patterns meets the temporal relation. Table 4.9 shows those seven of the 13 temporal relations defined by Allen [8] which are implemented in Asbru. The missing six relations are inverse to six of the implemented ones, i.e., they can be achieved by swapping the arguments.

The following subsections contain algorithms to monitor each of the relations shown in Table 4.9. In any case, the input for the monitoring algorithm consists of episodes represented by EpisodeDataPoints. Therefore, for each input channel there are four types of events: positive flank, negative flank, end of validity, and end of monitoring. Compare Table 4.10. Since the episodes that are the input to this abstraction may be found by a parameter proposition based on now, it is possible that one of the input episodes is revoked, because its end of validity has arrived. In this case, output already produce based on that episode must be revoked too, and any intermediate state based on that episode is reset to *Start*.

All instances of intervals A and B are assumed to start after the start of the state machines, which is easily implemented by inserting an *undefined* value as the very first value of every parameter as described for the matching of parameter propositions.

On the implementation level, the information that there will not be another interval often arrives together with the start or end of an interval. In this case, the state machines first process a negative flank event and then the *NoMore* event.

There can be multiple instances of A and B. For each matching pair a resulting

PF_A	An instance of interval A starts.
PF_B	An instance of interval B starts.
NF_A	The most recent instance of interval A ends.
NF_B	The most recent instance of interval B ends.
$NoMore_A$	There will not be another interval A.
$NoMore_B$	There will not be another interval B.
A revoked	End of validity for A has arrived.
B revoked	End of validity for B has arrived.

Table 4.10: Events processed when monitoring a temporal constraint.

output episode is produced. The start (positive flank) of each episode is the start of the first of the two episodes in the pair. The end (negative flank) is the end of the later of both episodes. In other words, the output episode envelopes both input episodes A and B.

4.5.4.1 A before B

A ends before B starts. There can be many pairs of instances of A and B for which this relation holds. Finding them is implemented by storing all complete occurrences of A and producing a set of pairs whenever an instance of B starts. When this instance of B ends, for each episode previously output NF must be added. When an instance of A or B is revoked, all episodes previously output must be revoked.

To this end the following records are kept.

Output-list. Whenever an episode is output, it is registered in the output-list by a record containing the following fields: the ID of the output episode (ID_{out}), the IDs of those instances of A and B referred to by this output episode (ID_A and ID_B), and the valid time of the positive flank (PF).

A-list. For each occurrence of A, PF and ID are stored.

Table 4.11 shows the pseudo code for monitoring this relation.

4.5.4.2 A overlaps B

A ends after B starts. Here the pairs of occurrences of A and B are formed by A's ending later then the corresponding B starts. Therefore all occurrences of B must be stored and whenever an occurrence of A ends, it forms a pair with each of the until then started occurrences of B.

Most of the previous occurrences of B already ended. For them, only one EpisodeDataPoint is output. Its PF_{out} is the earlier of PF_A and PF_B and NF_{out} is NF_A. One or – for overlapping occurrences of B – more occurrences of B may be open at the time an occurrence of A ends. For them, two EpisodeDataPoints are output, one at NF_A with undefined NF_{out} and one at NF_B with $NF_{out} = NF_B$.

The records necessary to produce this output are similar to the above case with the difference that here a list of occurrences of B is kept instead of those of A.

Event	Reaction
$\mathrm{NF_A}$	add this occurrence of A to the A-list
$\mathrm{PF_B}$	for each entry in A-list output a new episode with $\mathrm{NF_{out}}$ undefined and add it to output-list
$\mathrm{NF_B}$	for each entry in output-list where $\mathrm{ID_B}$ equals the ID of this occurrence report $\mathrm{NF_{out}} = \mathrm{NF_B}$ of the existing episode
A revoked	for each entry in output-list where $\mathrm{ID_A}$ equals the ID of the revoked occurrence revoke the resulting episode and remove it from output-list remove this occurrence of A from A-list
B revoked	for each entry in output-list where $\mathrm{ID_B}$ equals the ID of the revoked occurrence revoke the resulting episode and remove it from output-list
$\mathrm{NoMore_B}$	report the end of monitoring

Table 4.11: Monitoring the Allen relation *A before B*. $\mathrm{PF_A}$ and $\mathrm{NoMore_A}$ are ignored.

Output-list. This does not differ to the above case.

B-list. For each occurrence of B, PF and ID are stored.

Table 4.12 shows the pseudo code for monitoring this relation.

4.5.4.3 A starts B

The valid time of the starting point of a pair of occurrences of A and B must be equal. Therefore matching pairs is extremely restricted compared to the above cases. If $\mathrm{PF_A}$ and $\mathrm{PF_B}$ occur together, they form a new output episode. $\mathrm{NF_{out}}$ is the later of both $\mathrm{NF_A}$ and $\mathrm{NF_B}$. As above, the result is revoked if one of both input episodes are revoked.

To implement this, the following records are kept.

Output-list. This does not differ to the above case.

fitting-pair. A Boolean flag showing that $\mathrm{PF_A}$ and $\mathrm{PF_B}$ occurred together.

A-ended. A Boolean flag showing that $\mathrm{NF_A}$ occurred before $\mathrm{NF_B}$.

B-ended. A Boolean flag showing that $\mathrm{NF_B}$ occurred before $\mathrm{NF_A}$.

Table 4.13 shows the pseudo code for monitoring this relation.

4.5.4.4 A equals B

Both start and end of a pair of occurrences of A and B must be equal. This cannot be detected before they are over. Therefore, the algorithm is very simple. Still, a list of output episodes must be kept in order to revoke episodes if input episodes are revoked.

Event	Reaction
PF_B	add this occurrence of B to the B-list
NF_A	for each entry in B-list
	output a new episode with $PF_{out} = \min(PF_A, PF_B)$ and $NF_{out} = NF_A$
	and add it to output-list
NF_B	for each entry in output-list where ID_B equals the ID of this occurrence
	report $NF_{out} = NF_B$ of the existing episode
A revoked	for each entry in output-list where ID_A
	equals the ID of the revoked occurrence
	revoke the resulting episode and remove it from output-list
B revoked	for each entry in output-list where ID_B
	equals the ID of the revoked occurrence
	revoke the resulting episode and remove it from output-list
	remove this occurrence of B from B-list
$NoMore_A$	report the end of monitoring

Table 4.12: Monitoring the Allen relation *A overlaps B*. PF_A and NoMoreB are ignored.

Event	Reaction
PF_A and PF_B simultaneously	output a new episode with $PF_{out} = PF_A$
	and add it to output-list
	set A-ended := B-ended := false
	set fitting-pair := true
NF_A	if fitting-pair and B-ended and fitting-pair
	then report $NF_{out} = NF_A$ of the existing episode
	set fitting-pair := false
	else set A-ended := true
NF_B	if fitting-pair and A-ended
	then report $NF_{out} = NF_B$ of the existing episode
	set fitting-pair := false
	else set B-ended := true
A revoked	if there is an entry in output-list where ID_A
	equals the ID of the revoked occurrence
	then revoke the resulting episode and remove it from output-list
B revoked	if there is an entry in output-list where ID_B
	equals the ID of the revoked occurrence
	then revoke the resulting episode and remove it from output-list
$NoMore_A$	report the end of monitoring
$NoMore_B$	report the end of monitoring

Table 4.13: Monitoring the Allen relation *A starts B*. PF_A and PF_B are ignored if not occurring simultaneously.

Event	Reaction
$\mathrm{PF_A}$ and $\mathrm{PF_B}$ simultaneously	set started-simultaneously := true
$\mathrm{NF_A}$	set started-simultaneously := false
$\mathrm{NF_B}$	set started-simultaneously := false
$\mathrm{NF_A}$ and $\mathrm{NF_B}$ simultaneously	if started-simultaneously then report new episode with $\mathrm{PF_{out}} = \mathrm{PF_A}$ and $\mathrm{NF_{out}} = \mathrm{NF_A}$ and add it to output-list set started-simultaneously := false
A revoked	if there is an entry in output-list where $\mathrm{ID_A}$ equals the ID of the revoked occurrence then revoke the resulting episode and remove it from output-list
B revoked	if there is an entry in output-list where $\mathrm{ID_B}$ equals the ID of the revoked occurrence then revoke the resulting episode and remove it from output-list
$\mathrm{NoMore_A}$	report the end of monitoring
$\mathrm{NoMore_B}$	report the end of monitoring

Table 4.14: Monitoring the Allen relation *A equals B*. $\mathrm{PF_A}$ and $\mathrm{PF_B}$ are ignored if not occurring simultaneously.

Output-list. This does not differ to the above case.

started-simultaneously. A Boolean flag indicating the both A and B started simultaneously.

Table 4.14 shows the pseudo code for monitoring this relation.

4.5.4.5 A meets B

The end of A and the start of B occur simultaneously. This resembles the relation *starts* with the difference that it is the end of A which is considered. Therefore, $\mathrm{NF_{out}}$ is always $\mathrm{NF_B}$.

The following records are kept.

Output-list. This does not differ to the above case.

fitting-pair. A Boolean flag showing that $\mathrm{NF_A}$ and $\mathrm{PF_B}$ occurred with the same valid time.

Table 4.15 shows the pseudo code for monitoring this relation.

4.5.4.6 A during B

A starts after B starts and ends before B ends. This can be translated to the rule "If within an occurrence of B A ends and $\mathrm{PF_A}$ of this occurrence is latter than $\mathrm{PF_B}$, then

Event	Reaction
NF_A and PF_B simultaneously	output a new episode with $PF_{out} = PF_A$ and add it to output-list set fitting-pair := true
NF_B	if fitting-pair then report $NF_{out} = NF_B$ of the existing episode set fitting-pair := false
A revoked	if there is an entry in output-list where ID_A equals the ID of the revoked occurrence then revoke the resulting episode and remove it from output-list
B revoked	if there is an entry in output-list where ID_B equals the ID of the revoked occurrence then revoke the resulting episode and remove it from output-list
$NoMore_A$	report the end of monitoring
$NoMore_B$	report the end of monitoring

Table 4.15: Monitoring the Allen relation *A meets B*. NF_A is ignored if not occurring simultaneously with PF_B.

this is a fitting pair". PF_{out} is always PF_B. NF_{out} is always NF_B and therefore never known at the time of finding the fitting pair.

The following records are kept.

Output-list. This does not differ to the above case.

PF-B. The positive flank of the current occurrence of B, or $+\infty$ if there is no occurrence of B at this time.

Table 4.16 shows the pseudo code for monitoring this relation.

4.5.4.7 A finishes B

A and B end at the same time. In this case, the obligatory output-list is the only record needed. This is because at the moment of detecting a fitting pair, all the information is available and nothing is added later.

The following records are kept.

Output-list. This does not differ to the above case.

Table 4.17 shows the pseudo code for monitoring this relation.

4.5.5 Monitoring combinations of temporal patterns

Temporal patterns can be combined by the Boolean relations *and, or*, and *xor*. Also, the compliment of a temporal pattern can be created by *not*. In terms of abstraction algorithm, this means that the episodes matching two temporal patterns are combined using these operators as described in the following subsections.

Event	Reaction
$\mathrm{NF_A}$	if PF-B < $\mathrm{PF_A}$ then output a new episode with $\mathrm{PF_{out}} = \mathrm{PF_B}$ and add it to output-list
$\mathrm{PF_B}$	set PF-B := $\mathrm{PF_B}$
$\mathrm{NF_B}$	set PF-B := ∞ for each entry in output-list where $\mathrm{ID_B}$ equals the ID of this occurrence report $\mathrm{NF_{out}} = \mathrm{NF_B}$ of the existing episode
A revoked	if there is an entry in output-list where $\mathrm{ID_A}$ equals the ID of the revoked occurrence then revoke the resulting episode and remove it from output-list
B revoked	for each entry in output-list where $\mathrm{ID_B}$ equals the ID of the revoked occurrence revoke the resulting episode and remove it from output-list
$\mathrm{NoMore_A}$	report the end of monitoring
$\mathrm{NoMore_B}$	report the end of monitoring

Table 4.16: Monitoring the Allen relation *A during B*. $\mathrm{PF_A}$ is ignored.

Event	Reaction
$\mathrm{NF_A}$ and $\mathrm{NF_B}$ simultaneously	output a new episode with $\mathrm{PF_{out}} = \min(\mathrm{PF_A}, \mathrm{PF_B})$ and $\mathrm{NF_{out}} = \mathrm{NF_A}$ and add it to output-list
A revoked	if there is an entry in output-list where $\mathrm{ID_A}$ equals the ID of the revoked occurrence then revoke the resulting episode and remove it from output-list
B revoked	if there is an entry in output-list where $\mathrm{ID_B}$ equals the ID of the revoked occurrence then revoke the resulting episode and remove it from output-list
$\mathrm{NoMore_A}$	report the end of monitoring
$\mathrm{NoMore_B}$	report the end of monitoring

Table 4.17: Monitoring the Allen relation *A finishes B*. $\mathrm{PF_A}$ and $\mathrm{PF_B}$ are ignored.

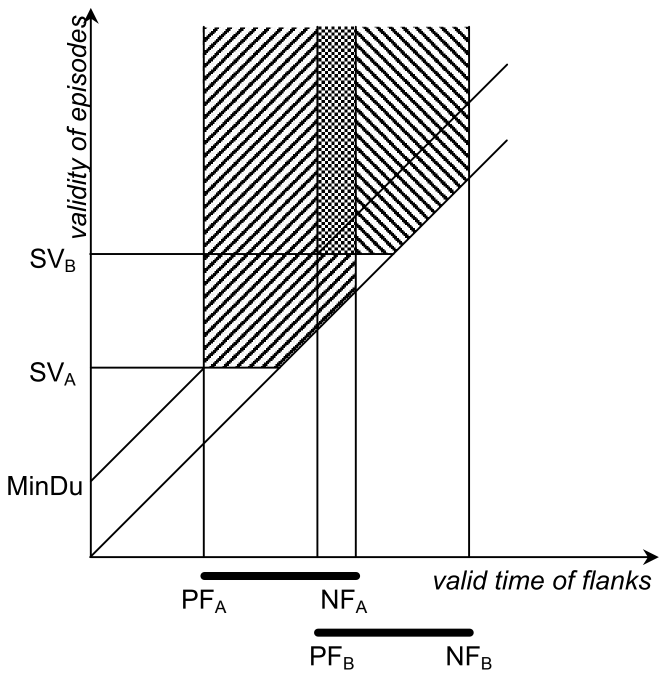


Figure 4.25: Defining the intersection of two episodes as their logical conjunction.

As for temporal constraints, the input can consist of episodes fitting a parameter proposition based on now in which case they are revoked if their end of validity arrives and thus the output based on them must be revoked, too.

While temporal constraints combine exactly two input channels, Boolean combinations take an arbitrary number of input channels, with the exception of *Not* which has exactly one input.

4.5.5.1 Boolean conjunction *and*

An episode is output whenever there is an intersection of both the interval of validity *and* the interval defined by PF and NF of episodes from all input channels.

Figures 4.25 and 4.26 illustrate this for two episodes. In Figure 4.25 the reference point is fixed and therefore the validity of the episodes does not end once they are found. Therefore, the validity of "A and B" starts when the second episode is found and it never ends.

In contrast, in the case shown in Figure 4.26 for reference point now, there is no episode of "A and B" since the validity of the episode of A ends before the start of validity of the episode of B. Of course, there will be other cases, where the intervals of validity overlap. Figure 4.26 just shows the extreme case to set it off against the case of the fixed reference point in Figure 4.25.

The following data structures are used in the process of monitoring a conjunction of episodes.

Output-list. Whenever an episode is output, it is registered in the output-list by a record containing the following fields: the ID of the output episode ($\mathrm{ID}_{\mathrm{out}}$) and

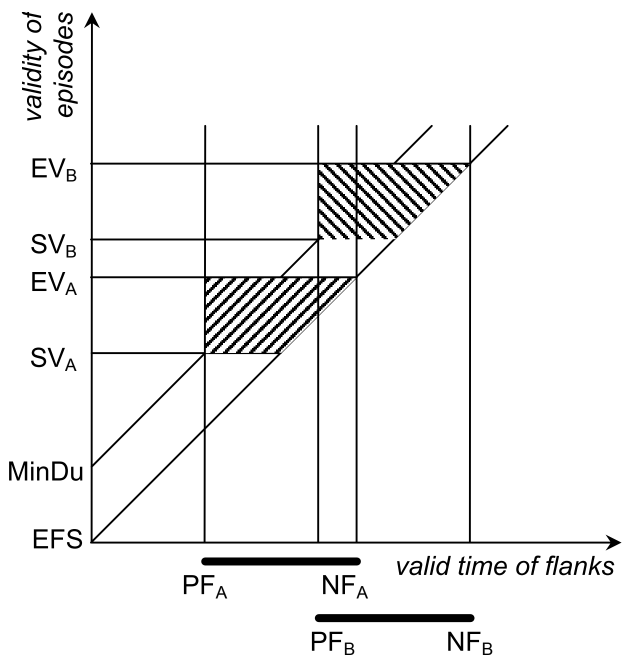


Figure 4.26: Two episodes having no intersection although PF of the second lies before NF of the first due to the delay in the matching process which is induced by MinDu and the restriction in validity imposed by EFS.

the IDs of those instances referred to by this output episode ($\mathrm{ID_I}$).

In. A list of Boolean variables indicating the current value of each of the inputs.

Out. A Boolean variable indicating the current status of the monitored combination.

PF. Most recent value of $\mathrm{PF_{out}}$.

Table 4.18 shows an algorithm to implement the conjunction of episodes. See Table 4.10 for an explanation of the events.

4.5.5.2 Boolean disjunction *or*

An episode is output whenever there is any episode from one of the input channels. This same episode is prolonged if another, overlapping episode in the input arrives and it is shrank or split if one of the episodes it is based on is revoked. Figure 4.27 shows the following sequence of events.

1. Episode A alone results in an equal output episode.
2. After PF of episode B the result is replaced by an episode which has undefined NF. Although we do not know, whether B will even stretch as far as A, we know that the resulting episode will not shrink. We could therefore wait until $\mathrm{NF_A}$ with replacing $\mathrm{NF_{out}}$ with *undefined.*
3. At NF of episode B the end of the resulting output episode is reported.

Event	Reaction
PF_i	set In_i := true if and(In_i) for all inputs i then output a new episode with $PF_{out} = PF_i$ and add it to output-list set Out := true

NF_i	set In_i := false
	if Out
	then report $NF_{out} = NF_i$ of the existing episode
	  set Out := false

i revoked	for each entry in output-list where ID_i equals the ID of the revoked occurrence revoke the resulting episode and remove it from output-list
$NoMore_i$	report the end of monitoring

Table 4.18: Monitoring the Boolean conjunction *and*.

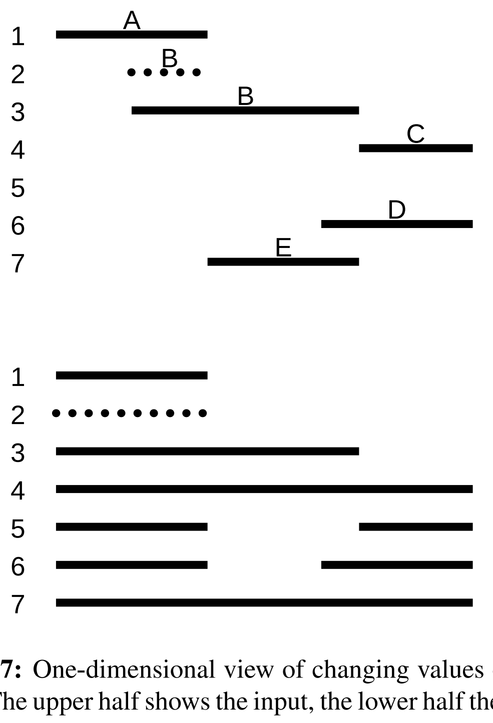


Figure 4.27: One-dimensional view of changing values of a disjunction. The upper half shows the input, the lower half the output. The numbers at the left show the temporal order.

4. After episode C is reported, the output episode is replaced by a longer one.
5. Revoking B leads to a split of the output episode.
6. Reporting episode D which starts before C leads to a prolongation of the second episode toward the past.
7. Episode E closes the gap in the output. The two episodes are replaced by one.

Note that at each step except step 3 a new instance of output episode is produced and the previous output is revoked.

The only modification of the flanks of an episode that other modules produce is the reporting of a previously unknown end. Neither start nor end are modified after they have been output for the first time. Replacing episodes with new ones which are shorter or longer on the other hand changes the count of episodes temporarily (between revoking the original one and outputting the new version). This can be handled by the abstraction modules receiving these episodes by processing all information at a given valid time and then producing output. This way, the count of episode does not change if one is replaced by another and the duration only changes in the appropriate way. Therefore, the module described here replaces episodes if they change their extent. The IDs of episodes are reused, i.e., the replacement episode has the same ID as its precursor.

To achieve this behaviour, the abstraction module must keep track of which input episodes contribute to which output episodes. Only those output episodes which must be changed should be revoked, the others should not be touched. To this end the following data structures are used.

Output-list. This is a list of output segments.

Output-segment. An output segment is a fraction of an output episode which is constituted by the same set of input episodes. I.e., whenever one of the inputs changes, a new output-segment is created. It contains the following fields: The start (PF_{seg}) and end (NF_{seg}) of the segment, the ID of the output episode (ID_{out}), and a list of IDs of input episodes (Contributors) which contributed to this segment of the output episode.

Gap. A Boolean flag indicating whether the revocation of an input episode cause a gap in the output just before.

to-report. List of episodes which must be output as new episodes after processing the current input.

to-revoke. List of episodes which must be revoked after processing of input and before reporting the new episodes in to-report.

From the example in Figure 4.27 the following complete list of cases can be constructed.

Case A. The complete input episode falls into a temporal region in which there previously was no output episode.

Case B. A previously existing episode is prolonged at the start.

Case C. A previously existing episode is prolonged at the end.

Case D. One or more gaps between existing episodes are closed.

Case E. The previously unknown NF of an existing episode is reported.

Case F. An existing episode is shortened at the start.

Case G. An existing episode is shortened at the end.

Case H. An existing episode is split in two episodes with a gap in between.

Case I. An existing episode is revoked completely.

Case J. Nothing changes in the result.

Cases A to D are reactions to $\mathrm{PF_i}$. Case E reacts to $\mathrm{NF_i}$. Cases F to I are reactions to revoking an input episode. Case J results from either $\mathrm{PF_i}$ of an interval completely lying within existing episodes or $\mathrm{NF_i}$ of an interval which in no part is the sole contributor to an existing episode.

The distinction between A-D+J and F-I lies in the amount of overlap between the changing episode and the other input episodes. Cases B-D do not exclude each other and so do F-H.

Tables 4.19 to 4.21 show the pseudo code for an implementation of the necessary segment management. For $\mathrm{PF_i}$ we distinguish (in the order it appears in the tables):

$\mathbf{A}_1$ The new episode lies completely before all previous ones with a gap in between.

$\mathbf{B}_1$ The first episode is prolonged at the start.

$\mathbf{A}_2$ The new episode lies past the last one with a gap in between.

$\mathbf{C}_1$ The last episode is prolonged at the end.

$\mathbf{J}_1$ The existing episode starts the new episode.

$\mathbf{J}_2$ The new episode includes the existing episode.

$\mathbf{J}_3$ The new episode finishes the existing episode.

$\mathbf{J}_4$ The existing episode includes the new period.

$\mathbf{D}_1$ The new episode closes a gap leading to a merge of previously distinct episodes.

$\mathbf{C}_2$ The previous episode is prolonged at the end.

$\mathbf{B}_2$ The next episode is prolonged at the start.

$\mathbf{A}_3$ The new episode lies in the middle of a gap.

For better overview, the cases J_1 to A_3 are depicted graphically in Figure 4.28.

4.5.5.3 Boolean disjunction *xor*

This is a variant of the above. Episodes are output for those periods of time, during which exactly one input episode is valid, i.e., if the number of contributors is exactly 1.

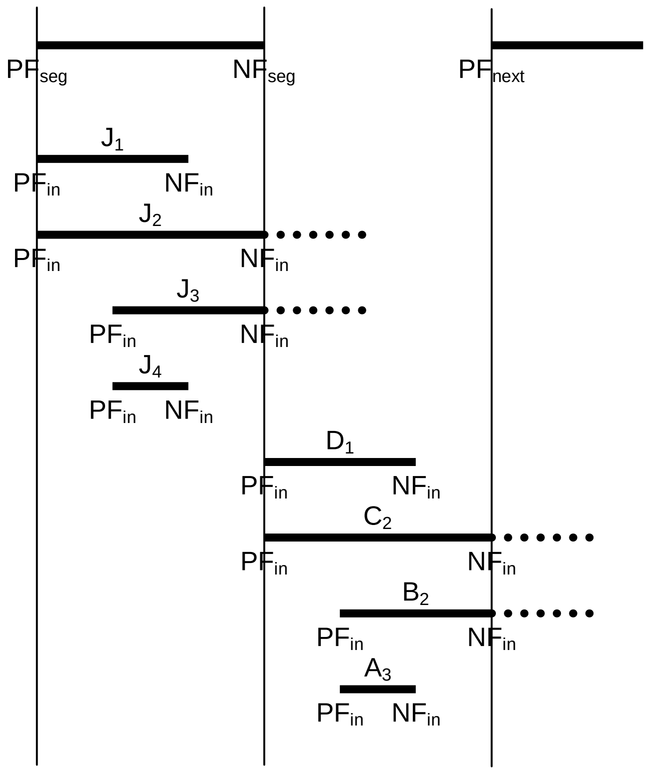


Figure 4.28: Cases J_1 to A_3. They are examined for each existing output-segment.

Event Case label	Reaction	Comment
PF_i	if $PF_i < PF_{first}$	Index $_{first}$ denotes the first output-segment
	then if $NF_i < PF_{first}$	in the chronologically sorted output-list
A_1	then insert new output-segment as first in output-list	
	with new ID, contributors = ID_i, PF = PF_i, NF = NF_i	
	and add the new ID to to-report	
B_1	else insert new output-segment as first in output-list	
	with ID=ID_{first}, contributors = ID_i, PF = PF_i, NF = NF_i	
	and add ID_{first} to to-report	
	if $NF_i > NF_{last}$	Index $_{last}$ denotes the last output-segment
A_2 and C_1	then append a new output segment similar to the above case	
	find the first output-segment for which $PF_i \geq PF_{seg}$	
	while $PF_i < NF_i$	The interval is shrunk as
	if $PF_i > NF_{seg}$	output segments are found
	then if $PF_i = PF_{seg}$	
	then if $NF_i \geq NF_{seg}$	
J_1	then add ID_i to $\text{contributors}_{seg}$	
	$PF_i := NF_{seg}$	
J_2		Index $_1$ denotes the first half
		of the duplicated segment,
		index $_2$ the second
	else duplicate this segment	
	$PF_2 := NF_1 := NF_i$	
	add ID_i to contributors_1	
	end loop	
	else if $NF_i \geq NF_{seg}$	
J_3	then duplicate this segment	
	$PF_2 := NF_1 := PF_i$	
	add ID_i to contributors_2	
	$PF_i := NF_{seg}$	
J_4	else triplicate this segment	
	$PF_2 := NF_1 := PF_i$	
	$PF_3 := NF_2 := NF_i$	
	add ID_i to contributors_2	
	end loop	

Table 4.19: Pseudo code for Boolean disjunction *or*, part 1.

Event Case label	Reaction	Comment
PF_i continued	if $NF_{seg} < PF_{next}$	there is a gap
	then if $PF_i = NF_{seg}$	
	then if $NF_i \geq PF_{next}$	
D_1	then insert new segment with $ID_{new} = ID_{seg}$	
	$PF_{new} := NF_{seg}$	
	$NF_{new} = PF_{next}$	
	add ID_i to $contributors_{new}$	
	and add ID_{seg} to to-report and to-revoke	
	replace all occurrences of ID_{next} by ID_{seg}	
	add ID_{next} to to-revoke	
	$PF_i := PF_{next}$	
C_2	else insert new segment with $ID_{new} = ID_{seg}$	
	$PF_{new} := NF_{seg}$	
	$NF_{new} := NF_i$	
	add ID_i to $contributors_{new}$	
	and add ID_{seg} to to-report and to-revoke	
	end loop	
	else if $NF_i \geq PF_{next}$	
B_2	then insert new segment with $ID_{new} = ID_{next}$	
	$NF_{new} := PF_{next}$	
	$PF_{new} := PF_i$	
	add ID_i to $contributors_{new}$	
	and add ID_{next} to to-report and to-revoke	
	$PF_i := PF_{next}$	
A_3	else insert new segment with new ID_{new}	
	$NF_{new} := NF_i$	
	$PF_{new} := PF_i$	
	add ID_i to $contributors_{new}$	
	and add ID_{new} to to-report	
	end loop	
	proceed with next segment	
NF_i	$NF_{last} := NF_i$	only for the last output segment NF
	report ID_{last}	can be undefined
ID_i revoked	for each output segment (in chronological order)	
	if ID_i is in $contributors_{seg}$	
	then remove ID_i from $contributors_{seg}$	
	if count of $contributors_{seg} = 0$	
	then if $ID_{seg} = ID_{previous}$ and $ID_{seg} = ID_{next}$	
	then replace ID_{seg} in all segments after this one by a new ID	
	add ID_{seg} and new ID to to-report	
	add ID_{seg} to to-revoke	
	else add ID_{seg} to to-report	

Table 4.20: Pseudo code for Boolean disjunction *or*, part 2.

NF_i	$NF_{last} := NF_i$ report ID_{last}	only for the last output segment NF can be undefined

reporting and revoking	performed after each processing of input for a certain valid time
	for each ID_{revoke} in to-revoke
	    output an EpisodeDataPoint with ID = ID_{revoke}
	    and both PF and NF undefined
	for each ID_{rep} in to-report
	    find first sequence with $ID_{seg} = ID_{rep}$ and set $PF_{rep} := PF_{seg}$
	    find last sequence with $ID_{seg} = ID_{rep}$ and set $NF_{rep} := NF_{seg}$
	    output an EpisodeDataPoint with ID_{rep}, PF_{rep}, and NF_{rep}	

Table 4.21: Pseudo code for Boolean disjunction *or*, part 3.

Event	Reaction	Comment
PF_i	if $NF_{last} < PF_i$ then report $NF_{out} = PF_i$ for previous episode store episode, even if $PF_{out} = NF_{out}$	
NF_i	report $PF_{out} = NF_i$ of a new episode with currently undefined NF	
in revoked	find $ID_{previous}$ for which $NF_{previous} = PF_i$ and revoke it find ID_{next} for which $PF_{next} = NF_i$ and revoke it create new episode with $PF_{out} = PF_{previous}$ and $NF_{out} = NF_{next}$	

Table 4.22: Pseudo code for Boolean negation *not*.

4.5.5.4 Boolean complement *not*

This abstraction takes exactly one stream of episodes as input. It produces an episode for each gap between the input episodes. The algorithm in Table 4.22 asserts that input episodes do not overlap.

The only data structure used by the algorithm is a list of output episode. For each episode, only the start and end are stored.

4.5.6 Monitoring count constraints

The temporal pattern in Asbru traditionally had a child element which specifies that a temporal pattern given as argument has to occur for a certain number of times. This is a special case of a constraint on the number of occurrences of a temporal pattern. It therefore is implemented by the Asgaard data abstraction unit by simply instantiating a count module (Section 4.5.7.3) and a comparison module (Section 4.2.3.6) to implement this Asbru element.

4.5.7 Extracting features of episodes

All the above abstractions produce output of type *EpisodeDataPoint*. I.e., each output item describes one or more interval(s) matching a certain time annotation resp. parameter proposition. The following modules extract single features about such episodes and supply them as numerical values to further abstraction modules.

These values can be used in further abstractions. However, their usage is limited by the fact that they always refer to the most recent episode and there is no way to refer to a certain instance in a series of episodes.

Also note that depending on the details of the time annotation, the detection of *matching* positive flanks can be severely delayed, e.g., if a minimum duration has to pass before the previous positive flank can be seen as belonging to an interval matching the time annotation.

In the remainder of this subsection, "at PF" refers to the time point at which a positive flank is found to be the positive flank of a matching interval, and not the time at which the positive flank occurs.

4.5.7.1 Episode as Boolean parameter

Output. The output is *true* during an episode and false otherwise. I.e., at PF a *true* value is output *with the valid time set to the positive flank* for each positive flank, and a *false* value is output with the valid time set to the negative flank for each negative flank.

Timing. Output is produced for each of the two flanks of each input episode. For PF, it cannot be produced before SV. For NF it is produced at NF (not EV).

4.5.7.2 State of fulfilledness as Boolean parameter

Output. The output is *true* when at least one episode is valid. It is *false* if none is valid and the end of monitoring is reached. This means that initially the output is the *undefined* value. At the first SV, a *true* is output. If the number of valid episodes drops to zero later, the *undefined* value is output. Only if no episode is valid and the end of monitoring is reached, the *false* value is output.

Timing. Output is produced at SV if previously no episode was valid and at EV if the corresponding episode was the only valid one; and at end of monitoring if no condition is fulfilled.

4.5.7.3 Count of episodes

Output. The number of valid (i.e. not yet revoked) episodes. It is increased at each SV and decreased at the end of validity (EV). PF and NF themselves do not change the count.

Timing. Output is produced whenever new input arrives.

4.5.7.4 Duration of episode

Output. The duration of the most recent episode. At PF this value is set to zero. Until NF, the output changes proportionally to the progress of valid time. Then the output stays unchanged. When the most recent interval is revoked, output is set to zero. If an episode before the most recent one is revoked, the output does not change.

Arguments. The time between two output data points for the time during which the duration changes continually.

Timing. Output is produced a) at PF, b) between PF and NF at the rate specified by the argument, c) at NF, and d) at the end of validity of the most recent interval.

4.5.7.5 Total duration of episodes

Output. The sum of all durations of currently valid episodes. This is similar to the above, with two differences: First, the value is not reset at SV but further increased. Second, revoking any episode leads to a reduction of the output value by the duration of the revoked episode.

Arguments. The time between two output data points for the time during which the duration changes continually.

Timing. Output is produced a) from PF to NF at the rate specified by the argument, and b) at the end of validity of any episode.

4.5.7.6 Start of episode

Output. The valid time of the positive flank of the most recent episode. It this episode is revoked, the undefined value is output.

Timing. Output is produced at each positive flank and when the most recent episode is revoked.

4.5.7.7 End of episode

Output. The valid time of the negative flank of the most recent episode. It this episode is revoked, the undefined value is output. Also, at the positive flank the undefined value is output since between the positive and the negative flank the end of the current episode is not known.

Timing. Output is produced at each flank and when the most recent episode is revoked.

4.6 Integration of plan execution

The following first discusses the life cycle of Asgaard plans in detail. This forms the basis for the modules implementing the various forms of Asbru plans and plan steps described thereafter. Here, we focus on the principal mechanisms of plan execution.

4.6.1 Asbru plan semantics

Each plan traverses a graph of plan states during its life cycle. The transition between these states is controlled by conditions as shown in Figure 4.29. There are three modes of traversing from one state to another.

1. In the default case, the plan state changes without user interaction as soon as the condition is fulfilled.

2. If the flag *confirmation-required* is set to *yes*, the plan state does not change automatically. Instead, a request to the user interface is sent, as soon as the condition is fulfilled. When the user confirms the state transition, then it is enacted. If the condition becomes no more fulfilled, before the user confirmed the state transition, then the request to the user interface is cancelled.

3. If the flag *overridable* is set to *yes*, the user is free to enact the plan state transition independent of the condition being fulfilled or not. For such conditions, a message is sent to the user interface as soon as the plan enters a state in which the corresponding condition is relevant. If the plan leaves this state, the original message is cancelled.

   When a condition with flag *overridable* becomes fulfilled based on its inputs, then it is treated as if the flag was not there. I.e., depending on the value of flag *confirmation-required*, case 1 or 2 above are considered.

4.6.2 Principal design of plan state modules

The principal design of plan state modules is similar to data abstraction modules. The most significant difference is the complex handling of the inputs.

4.6.2.1 Inputs

Each *PlanStateModule* has the following inputs:

- filter-precondition
- setup-precondition
- suspend-condition
- reactivate-condition
- complete-condition
- abort-from-suspended-condition
- abort-from-activated-condition
- synchronization-input
- parent-plan-state
- child-plan-state

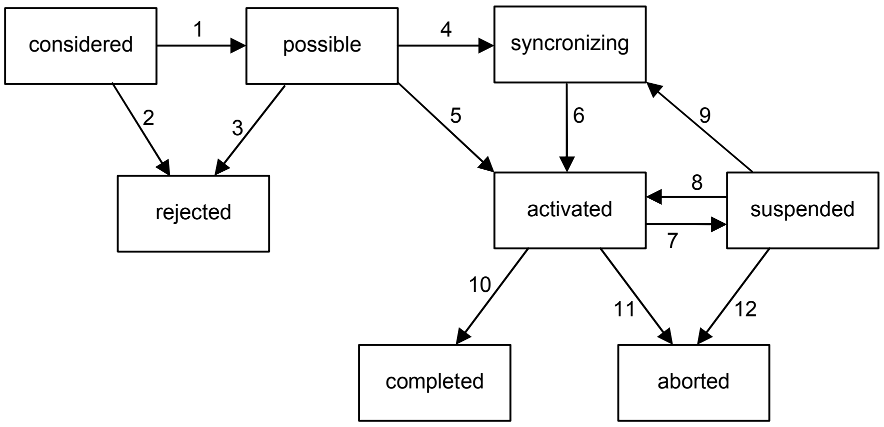


Figure 4.29: Asbru plan states. State transitions occur under the following conditions.
1 filter-precondition fulfilled;
2 filter-precondition never more fulfilled;
3 setup-precondition never more fulfilled;
4 setup-precondition fulfilled and child of parallel or any-order plan;
5 setup-precondition fulfilled and child of sequential or unordered plan;
6 sole child of an any-order parent plan, or all children of a parallel parent plan;
7 supend-condition fulfilled;
8 reactivate-condition fulfilled and not child of an any-order plan;
9 reactivate-condition fulfilled and child of an any-order plan;
10 complete-condition fulfilled and continuation-specification fulfilled;
11 abort-condition fulfilled or propagation-specification fulfilled;
12 as previous item, but a different abort-condition can be given for suspended state.
For all transitions but 2, 3, and 6, the following two rules apply:
If the flag *overridable* is set, then the user can cause this state transition at any time.
If the flag *confirmation-required* is set, then the transition does not take place before the user confirms it.

- earliest-starting-time
- latest-starting-time
- earliest-finishing-time
- latest-finishing-time
- minimum-duration
- maximum-duration

The type of the inputs ending on "condition" is EpisodeDataPoint. Each of these inputs is connected to the output of a module which implements the corresponding condition, e.g., a ParameterPropositionModule. The mapping from Asbru to abstraction modules is discussed in 4.7.3. If the *abort-condition* does not differentiate between the states suspended and activated, then both inputs *abort-from-*-condition* are connected to the same module.

The type of *synchronization-input* is IntegerDataPoint. This input is connected to one of the *synchronization-outputs* of the parent plan. Only two data points are transferred through the normal lifetime of a plan via this channel. When the plan is logically instantiated, the parent sends the symbolic value *start* to the child plan. If the argument *synchronize* is true, then the parent sends the symbolic value *activate* to this plan causing it to change from plan state synchronized to activated.

The type of inputs ending on "plan-state" is PlanStateDataPoint. In addition to the valid time, this "data point" contains the new plan state and one synchronization flag per child plan, called *synchronization-outputs* here. The *parent-plan-state* is connected to output *my-plan-state* of the parent plan. The *child-plan-state* is connected to *parent-plan-state* of all child plans. See further below for the propagation of plan states between parent and child plans.

The type of the inputs ending on "time" is DateDataPoint. These inputs are connected to modules performing the calculation of these time points, if specified.

The type of the inputs ending on "duration" is TimeDataPoint. These two inputs are connected to similar modules directly mapping the expression given in the Asbru element *time-annotation* in *plan-activation*. Compare Section 4.7.5.1.

4.6.2.2 Output

Each PlanStateModule outputs PlanStateDataPoints. This "data point" contains the new plan state, called my-plan-state here; an array of synchronization flags, one per child plan; and an optional numeric value.

This PlanStateDataPoint is sent to three types of receivers. The parent plan only reads the plan state of the child. The child plans read the plan state of the parent and the synchronize flag relating to them. Various abstraction modules read the numeric value.

4.6.2.3 Arguments

Each *PlanStateModule* receives a symbolic value named *synchronizationmode* as an argument when it is created. It has one of four values.

none. The *PlanStateModule* directly changes from states *possible* and *suspended* to state *activated* when the corresponding condition is fulfilled. This is used for sequential and unordered subplans which do not have an *on-suspend* plan defined in their plan activation.

start. The *PlanStateModule* does not directly change from state *possible* to *activated*. Instead, it enters state *synchronizing*. It remains in this state, until the parent sends a data point with the value *activate* to the *synchronization-input*. However, the transition from *suspended* to state *activated* occurs instantaneously. This is used for parallel subplans.

both. Both the transitions from states *possible* and *suspended* to state *activated* are delayed until the parent sends a data point with the value *activate* to the *synchronization-input*. In both cases, the state *synchronizing* is entered while waiting[8]. This is used for *any-order* subplans.

resume. The *PlanStateModule* does not directly change from state *suspended* to *activated*. Instead, it enters state *synchronizing* and remains in this state, until the parent sends a data point with the value *activate* to the *synchronization-input*. This is used for plans which have an *on-suspend* plan defined in their plan activation.

4.6.2.4 Definition and timing of successful termination

An Asbru plan terminates successful, i.e., it is *completed*, if *all* of the following conditions are true:

- Its *complete-condition* is fulfilled.
- The child plans specified in the *continuation specification* completed.
- The current time lies in the interval between the *Earliest Finishing Time* and the *Latest Finishing Time* prescribed by the time annotation for this plan.[9]

The PlanStateModule builds an internal representation of the *continuation specification* when it is created. The *continuation specification* is given by the Asbru element *wait-for*. It is either a logical expression based on plan references or a count. In case of a logical expression, each leaf of the expression tree is internally represented by a Boolean which is true if the referenced plan already completed. This expression is evaluated in certain cases described below.

If only the count of child plans which must complete before the parent can complete is given (as stated by *one*, *cardinality*, or *all*), then only this count is maintained.[10]

[8]This means that there is no consequence if the suspend condition is no more fulfilled while still waiting for the parent to allow the reactivation.

[9]Compare 4.5.1 for details on mapping a time annotation to absolute time points for earliest and latest starting and finishing time.

[10]In the rare case, in which the same child plan completes more than once, each time is counted. This does not occur for *retry-aborted-subplans*, but only if a plan is listed more than once in the list of children, e.g., do first A, then B, then A again, in a sequential plan.

4.6.2.5 Definition and timing of failure

An Asbru plan has two modes of failure: It can be *rejected* or *aborted*. A plan is rejected if *one* of the following is true:

- The filter-precondition can never more be fulfilled, and the plan is still in considered state.
- The setup-precondition can never more be fulfilled.[11]
- The Latest Starting Time arrived and the plan is still in state *considered* or *possible*.

A plan is aborted if *one* of the following is true:

- The abort-condition is fulfilled.
- The *propagation specification* evaluates to true.
- The Latest Finishing Time arrived and the plan is in state *synchronizing*, *activated*, or *suspended*.

Furthermore, the plan is terminated if the parent plan aborts. In this case, it is not important, whether the plan is aborted, completed or rejected.

The internal representation of the *propagation specification* differs from that for the *continuation specification* insofar, as it distinguishes between rejected and aborted subplans. Therefore, each leaf of the expression tree consists of a pair of Booleans. One Boolean is true, if rejected is propagated, the other if aborted is propagated.[12] A third Boolean for each leaf is set to true if this plan was rejected and reject is propagated or if this plan was aborted and abort is propagated. It is this flag which is combined with those for the other leafs to evaluate the status of the *propagation specification*.

If the *propagation specification* is not given, the *continuation specification* is translated to a similar *propagation specification*. This is done by inverting the Boolean operators and setting both Booleans (reject and abort propagation) for each plan which is mentioned in the *continuation specification*.

There is no cardinality constraint in the *propagation specification* in Asbru, but the inversion of a *continuation specification* demanding that x out of N subplans complete means that a plan is aborted if $N - x + 1$ subplans fail, where fail means to either be aborted or rejected.

If the flag *retry-aborted-subplans* is set, then the failure of a subplan is not propagated to the parent before either the time annotation for that subplan does not allow completion anymore[13], or the filter-precondition or the setup-precondition become never more fulfilled.

[11]This is independent from attempts of the plan execution unit to start plans which lead to fulfilling the setup-condition. If the setup-precondition is *never more fulfilled*, these attempts obviously had failed.

[12]"rejected is propagated" is short for "the rejection of the subplan can lead to aborting the parent plan", i.e., the parent can never be rejected because subplans of it were.

[13]There are two cases: The plan is *considered* or *possible* and the Latest Starting Time is passed; or the plan is *synchronizing*, *suspended*, or *activated* and the Latest Finishing Time is passed.

	considered,	rejected	synchronizing,	suspended	aborted,
	possible		activated		completed
activated	+	+	+	+	+
suspended	+	+		+	+
aborted		+			+
completed		+			+

Table 4.23: Possible child plan states for a given parent plan state. Parent plan states are shown in the first column. A plus means that the child plan may be in the state named shown in the first line of the table.

4.6.2.6 Parent-child relations in plans

Table 4.23 shows which plan states of a child are possible for each of the plan states of the parent plan. During the plan states which are omitted in the table (*considered*, *possible*, and *synchronizing*), child plans do not exist.

Tables 4.24 to 4.26 show the reactions to all possible types of input or events for each of the plan states considered, possible, synchronizing, activated, and suspended. Plan states aborted and completed are terminal, no state transitions occur after one of these states is reached. In tables 4.25 and 4.26, the word *depending* stands for the plan states *completed* and *aborted*, depending on the evaluation of the *complete-specification* or *propagation-specification* at the left on the same row.

Initially, when all plan state modules are created, they are in internal state *uninstantiated.* When the parent instantiates a child, it sends the symbolic value *start* through the synchronization connection to that child which then enters the *considered* state.

The following miscellaneous issues need to be considered.

- If a plan activation contains an on-abort-plan (a plan to be performed instead of the original one if the original one is aborted), then that plan replaces the original one in the network of plan-state propagation, as soon as the original plan aborts. This means that the corresponding *child-plan-state* input of the parent plan is connected to the *my-plan-state* output of the on-abort-plan. Only the success or failure of the latter plan is propagated to the parent (and only if the plan activation of the on-abort-plan does not contain an on-abort-plan itself).

- If a plan is waiting for optional subplans to complete when the Latest Finishing Time arrives, it is aborted.

- The state *suspended* is entered if either the parent plan is suspended or the suspend-condition is fulfilled. Each of the two cases has its own trigger for the end. Both modes of being suspended can overlap. Therefore, they are modeled by two independent Booleans in Table 4.26: *parent-suspended* and *self-suspended.*

- When a plan is suspended, and an on-suspend-plan is defined in the plan activation, then this plan is started. See Section 4.6.3.6 for the timing of resuming the suspended plan.

current state	event	condition	next state
considered	filter-condition fulfilled		possible
	LST passed		rejected
	filter-condition cannot be fulfilled		rejected
	parent completed		aborted
	parent aborted		aborted
	parent suspended		considered
possible	EST passed and setup-condition fulfilled	synchronize = true	synchronizing
	EST passed and setup-condition fulfilled	synchronize = false	activated
	LST passed		rejected
	setup-condition cannot be fulfilled		rejected
	parent completed		aborted
	parent aborted		aborted
	parent suspended		possible
synchronizing	synchronize input from parent		activated

Table 4.24: Plan state changes for the selection phase.

- If the parent terminates (i.e., it is *completed* or *aborted*), the state of the child plan is not important. For simplicity, it is considered *aborted*.

4.6.3 Types of plans

This subsection describes actions carried out to execute each of the plan body types defined in Asbru. They can be nested in Asbru but nesting need not be discussed here since the Asbru parser handles the nesting when instantiating the suitable plan state modules. Also the suitable connections between the PlanStateModules are created by the Asbru parser.

In the following, only the behaviour in addition to that described above is mentioned. I.e., in each of the cases below, the modules behave as described in Figures 4.24 to 4.27.

4.6.3.1 Sequential Plans

As soon as state *activated* is entered for the first time[14], for each of the subplans the following steps are taken.

- The symbolic value *initialize* is sent to the PlanStateModule representing the subplan.

[14]The state *activated* is also entered after the reactivate condition is fulfilled if the plan had been *suspended* before.

event	condition	reaction	next state
suspend condition fulfilled		self-suspended := true	suspended
abort condition fulfilled			aborted
propagation-specification satisfied			aborted
LFT arrived			aborted
parent aborted			aborted
parent suspended		parent-suspended := true	suspended
subplan aborted	RAS = true	restart subplan	activated
subplan aborted	RAS = false and OAP exists	initiate on-abort-plan	activated
subplan aborted	RAS = false and no OAP exists	update propagation-specification	*depending*
subplan rejected		update propagation-specification	*depending*
complete condition fulfilled and continuation-specification satisfied and EFT passed			completed
child completed		update complete-specification take next plan step (for sequential and any-order plans)	*depending*

Table 4.25: Plan state changes for *activated* plans. Restarting a child plan is performed by sending *start* to its *synchronization input*.
depending stands for the plan states *completed* and *aborted*, depending on the evaluation of the *complete-specification* or *propagation-specification* at the left of the corresponding cell.
RAS stands for retry-aborted-subplans.
OAP stands for on-abort-plan.

event	condition	reaction	next state
reactivate condition fulfilled	parent-suspended = false	self-suspended := false	activated
reactivate condition fulfilled	parent-suspended = true	self-suspended := false	suspended
abort condition fulfilled			aborted
propagation-specification satisfied			aborted
LFT arrived			aborted
parent aborted			aborted
parent suspended		parent-suspended := true	suspended
suspend condition fulfilled		self-suspended := true	suspended
parent reactivated	self-suspended = true	parent-suspended := false	suspended
parent reactivated	self-suspended = false		activated
subplan aborted	RAS = true	restart subplan	
subplan aborted	RAS = false and OAP exists	initiate on-abort-plan	suspended
subplan aborted	RAS = false and no OAP exists	update propagation-specification	*depending*
subplan rejected		update propagation-specification	*depending*

Table 4.26: Plan state changes for *suspended* plans.
RAS stands for retry-aborted-subplans.
OAP stands for on-abort-plan.

event	reaction	next state
start	initiate conditions	considered

Table 4.27: Plan state changes for *uninstantiated*, *completed*, and *aborted* plans.

- When that module sends a PlanStateDataPoint with the value *completed*, then loop proceeds to the next subplan.
- The same holds if the value *aborted* is received from the subplan and *retry-aborted-subplans* is *false*.
- If *aborted* is received and *retry-aborted-subplans* is *true*, then the same subplan which sent this PlanStateDataPoint receives another *initialize* signal.

4.6.3.2 Parallel Plans

Here, all subplans enter the preselection phase and the active phase at the same time.

- When the state *activated* is entered for the first time, all subplans receive the *initialize* signal.
- When all subplans sent their *ready* signal, then they all receive the *activate* signal.
- When one of the subplans is aborted and *retry-aborted-subplans* is *true*, then this subplan immediately receives another *initialize* signal.

4.6.3.3 Any-Order Plans

Here, only one subplan can be active at a time, but the order in which they are performed is not fixed. Subplans which are suspended cannot be reactivated before the currently activated plan leaves this state. However, they are resumed before other plans (which were not started) are activated. To this end, two waiting lists are kept, one for those subplans to resume and one for those to activate.

- When the state *activated* is entered for the first time, all subplans receive the *initialize* signal.
- When a subplan sends its *ready* signal, it is added to the waiting list *subplans waiting to be activated*. If no subplan is activated, the first subplan from this waiting list is activated.
- When a subplan leaves the *activated* state, another subplan is activated, if possible. First, subplans on the waiting list *subplans waiting to be resumed* are considered, then the *subplans waiting to be activated*. If both list are empty, no subplan is activated until a subplan sends its *ready* signal.
- When the reactivate condition of a subplan is fulfilled, it does not change to *activated* immediately. Instead it sends the signal *ready to resume* to the parent plan. If another subplan is active, the parent then adds the suspended subplan to the list of *subplans waiting to be resumed*. Otherwise, this subplan receives the *activate* signal immediately.

4.6.3.4 Unordered Plans

When the state *activated* is entered for the first time, all subplans receive their *initialize* signal. No further synchronization is performed.

4.6.3.5 User-Performed Plans

When the state *activated* is entered, no further action is taken. Software interpreting the logfiles which show all plan state changes can interpret the activation of user-performed plan as issuing a certain recommendation.

4.6.3.6 Plan Activation

This module is instantiated if an Asbru *plan-activation* contains at least one of the following: plan arguments, return value assignment, on-abort plan activation, on-suspend plan activation. If none of then are present, then the PlanStateModule representing the Asbru parent plan and that representing the Asbru child plan are directly connected. Otherwise, a PlanActivationModule is inserted between them, i.e., it is the child of the module representing the Asbru parent plan and and the parent plan module of that representing the Asbru child plan.

This module has the following PlanStateModules associated as subplans:

- One AssignmentModule per argument of the Asbru plan.
- A PlanStateModule representing the Asbru plan itself.
- Optionally a PlanStateModule representing the *on-abort* plan.
- Optionally a PlanStateModule representing the *on-suspend* plan.
- One AssignmentModule per return value of the Asbru plan.

The following steps are taken by the PlanActivationModule.

- When this module enters the state *activated* for the first time, it sends all AssignmentModules representing argument assignments the *initiate* signal. As a result, these modules forward their input to the PlanModifiedParameterModules representing these arguments. Then they report their completion to the parent plan module (the module described here).
- Then, the PlanStateModule representing the Asbru plan itself receives the *initiate* signal.
- If that module's state becomes *aborted* and there is an *on-abort* plan, then this plan module receives the *initiate* signal. Its success is treated as the success of the original plan, but suspending the *on-abort* plan does not trigger the *on-suspend* plan.
- If the PlanStateModule representing the Asbru plan itself is suspended because of the suspend condition of that plan, the PlanStateModule representing the *on-suspend* plan receives the *initiate* signal[15].
- If an *on-suspend* plan is active and the reactivate condition of the original plan is fulfilled, the PlanStateModule representing it sends a *ready to resume* signal to the parent (the module described here). Only after the *on-suspend* plan leaves the *activated* state, the PlanStateModule representing the waiting plan receives an *activate* signal.

[15]If the parent plan suspends, the child plans are suspended, too, but their *on-suspend* plans are not started, because this would contradict the suspended state of the parent.

- If the *on-suspend* plan is aborted, then this is treated as if the original plan was aborted.

- If either the original plan, the *on-suspend* plan or the *on-abort* plan complete, then all AssignmentModules representing return value assignments receive the *initiate* signal. After these subplans completed (which occurs at the next time steps), the PlanActivationModule also completes.

- If the *on-abort* plan aborts or the original plan aborts and there is no *on-abort* plan, or if the *on-suspend* plan aborts and there is no *on-abort* plan, then the PlanActivationModule also aborts.

4.6.3.7 Assignment

This is instantiated for the Asbru elements *variable-assignment*, *parameter-assignment*, and *set-context*. It is also instantiated as part of the technical plans which are created on plan activation as described above.

Compared to other modules implementing plan steps, this module has an additional input: the source of the value it assigns. This is the output from an abstraction module (or the interface to file input, if the value is raw data from a file). This value is copied to the *data output* when the state *activated* is entered. Then the state *completed* is entered.

All of these activities take place it the same time step. The PlanStateModuleOutputDataPoint reporting both the completion of this module and the new value for the variable etc. have a valid time of one time step later than the current one.

4.6.3.8 If-Then-Else

This module is connected to a ComparisonModule which implements the *simple-condition* in the Asbru element. If this input has the value *true*, then the first child receives the *initiate* signal, otherwise the second one. The Asbru parser ensures that the first child is the *then* branch of the Asbru element and the second is the *else* branch. If more than one single step is contained in a branch, then the Asbru parser creates a technical plan to contain them (in sequential order). The final state of the selected subplan is propagated to the parent plan of the IfThenElseModule.

4.6.3.9 Ask

This module receives a reference to the parameter which is asked for and the timeout from the *ask* statement (measured in internal time steps) when it is created. Upon creation the output of the module representing that parameter is connected to the AskModule.

When the AskModule receives the *activate* signal, it adds the timeout to the current valid time and sends a request to the user interface for the given parameter until the calculated time point. Then sets a *postalarm* to the time point of the timeout.

If the parameter is entered before the timeout, the AskModule receives the new value as its input. It ignores the value – only the fact that the value was entered is important. It then cancels the alarm and completes.

If the parameter is not entered in time, the AskModule receives the previously set *postalarm* from the ManagementModule. In response, it completes. The current definition of the Asbru semantics does not include the possibility to abort for *ask*.

4.6.3.10 Cyclical Plans

Cyclical plans in Asbru have a rich assortment of ways to specify how often the repeated action is taken. Consequently, the module implementing them is connected to the following set of inputs.

- Time points *start-time* and *end-time*. These inputs are connected to modules implementing the calculation of the values given in the Asbru code, e.g., ConstantModule for constants, or AssignmentModule for variables. In both cases, only one data point is read when the cyclical plan enters active state.
- A group of integer inputs fed by similar modules implementing the limits set forth in Asbru elements *maximum-attempts* and *times-completed*. Again, they are only read on activation. Asbru element *maximum-attempts* actually implements six different failure counters. Accordingly, this element is mapped to six different numeric inputs. They are: *rejected-between-completions*, *aborted-between-completions*, *failure-between-completions*, *rejected-in-total*, *aborted-in-total*, and *failures-in-total*.
- A RepeatedTimePointModule which sends a dummy time point whenever a new instance of the repeated time point specified in Asbru element *set-of-cyclical-time-points* has arrived. If this element is not used in the Asbru code, the repeated time point is ignored, i.e., no data points are used from this input channel.
- Six values representing the ESS, LSS, EFS, LFS, MinDu, and MaxDu given in the *time-range*. These values are only read once, when the cyclical plan is activated. If they are all *undefined*, then there is not *cyclical-time-annotation* in the Asbru code and the repeated action is started without waiting for a new data point for the repeated time point.
- *Minimum* and *maximum* for *duration* and *retry-delay*, if given. Otherwise, these inputs are connected to the constant *undefined*.
- A ParameterPropositionModule implementing the *until-condition*. Again, only the constant *undefined* is received if no *until-condition* is given.

When the CyclicalPlanModule receives the *initialize* signal from its parent plan, it first checks whether a start time for the first repetition has been given, i.e., whether input *start-time* is undefined. If a value is given, then a timer (prealarm) is set for this time and only then the child plan receives this signal.

If no start time is given, the module checks whether all six input representing the time range are connected to the constant *undefined*. If they are, then there are no time constraints on the child plan invocation and the first instance of the repeated action is carried out immediately, which means that the child plan implementing it receives the *initialize* signal.

If at least one of the six values of the *time-range* is given, then a *cyclical-time-annotation* is specified, which will constrain these invocations. This means that before each invocation of the repeated action, the CyclicalPlanModule awaits at data point from the RepeatedTimePointModule, then adds the earliest starting shift to the received time point, and sets a timer (prealarm) for this date. Only after this alarm triggered (and all other conditions are fulfilled), the child plan receives the *initialize* signal.

If a minimum duration is given, the CyclicalPlanModule sets a timer accordingly before activating the child plan and only after this alarm was received and the child plan terminated, a new iteration is considered. The same method is used to implement waiting for the earliest finishing time, if the cyclical time annotation specifies it.

If a maximum duration is given, then the child plan is aborted if it did not complete before this period is elapsed. Likewise a latest finishing time is treated, if given.

If the child plan does not progress from *considered* state to *activated* state before latest starting time arrived or maximum retry delay elapsed, then the child plan is aborted. The further procedure is as if the child plan would have been activated.

If a minimum retry delay is given, a similar timer is set accordingly on termination of the child plan.

If a maximum retry delay is given, another timer is set. When this alarm is reached before the child plan entered active state, then it is terminated and it counts with the unsuccessful completions.

If an *end-time* is defined, i.e., if the corresponding input is not *undefined*, then another timer, in this case a postalarm, is set for this time. When the alarm is triggered, the cyclical plan is completed irrespective of the state of the current child plan iteration.

The same form of completion is performed if either a positive flank is read from the *until-condition* input.

The CyclicalPlanModule maintains five counters for the child iterations:

1. Rejections between completions, which is reset at each successful completion and counts rejected child plans.
2. Abortions between completions, which is reset at each successful completion and counts aborted child plans.
3. Total rejections, which is not reset and counts rejected child plans.
4. Total abortions, which is not reset and counts aborted child plans.
5. Total completions, which is also not reset and counts completed child plans.

failure-between-completions maps to the sum of the first two counters; *failures-in-total* to the sum of counters 3 and 4. If the child plan fails and one of the limits set forth in *maximal-attempts* is already reached, then the CyclicalPlanModule aborts. If the child plan completes and the total count of completions is reached, and the complete condition is fulfilled, if given, then the CyclicalPlanModule completes.

To obtain cyclical reference points, a helper module, the RepeatedTimePointModule, is needed. It receives the time point at which its operation starts, and the frequency, as inputs. The first is calculated by adding the content of Asbru elements *time-point* and *offset*. The latter is directly given in the Asbru element *frequency*. This module sends a dummy data point whenever a new time point in the series has arrived. Only the valid time of this time point is important.

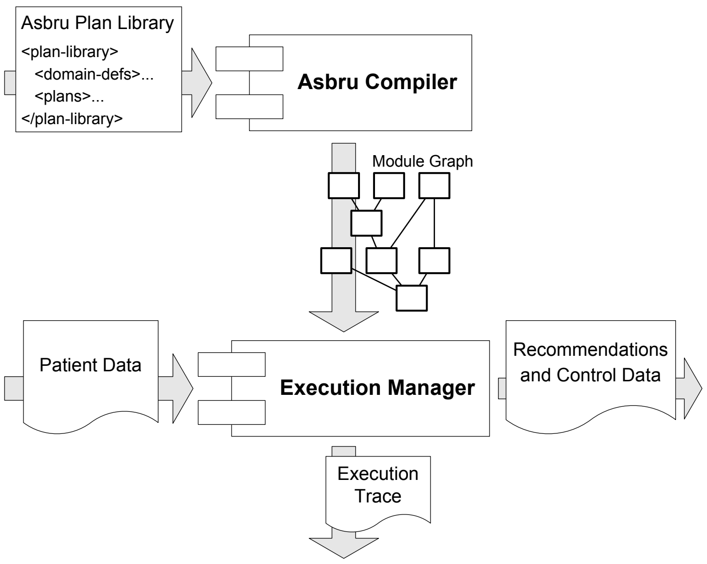


Figure 4.30: System architecture of the interpreter. The Asbru plan library is translated to a graph of modules by the Asbru compiler. This graph is fed with patient data by the execution manager, producing an execution trace on the one hand, and recommendations and control data on the other hand [131].

4.7 Bridge to Asbru

This section describes how the Asbru elements map to modules described in the sections above. A narrative line is followed. In contrast to the previous sections, we can follow the more intuitive top-down approach, starting at plans and breaking them down into their parts and subparts.

A full description of Asbru can be found in the Asbru Reference Manual [125]. This thesis section focuses on the mapping of the language elements to the modules described before. Therefore, only those parts of the language descriptions, which are indispensable for the understanding are given here. For further details on the usage of each of the language elements, the reader is referred to the Reference Manual.

The basic design is shown in Figure 4.30. The whole execution system is composed by the Asbru Compiler, the module graph, and the Execution Manager. The Compiler instantiates modules described in the previous sections to depict the Asbru Plan Library. The inputs and outputs of these modules are connected to form a graph. The data flow through this graph is mediated by the Execution Manager as described in Section 4.1.3. It also feeds patient data into the graph to trigger abstraction and plan execution.

During this process, an extensive Execution Trace is created to document the flow of information between the individual modules. Some modules output recommendations or control data, which is used by the environment in an implementation-dependent way, e.g., as input for user interfaces, or to control devices.

A plan library consists of definitions and plans. The definitions are discussed in section 4.7.7. The following starts with plans and goes top-down to definitions via conditions and parameter propositions.

Asbru has some elements not related to plan execution. They are simply skipped in this description.

There are a few features in Asbru 7.4 which were considered interesting when version 7 was designed in 2000, but never saw a use case in the decade following. Some of them are excluded from the discussion in this thesis, most notably complex data structures. See Section 5.4.7 for details.

4.7.1 Plans

Each plan contains conditions (described in Section 4.7.3), a plan body (described in Section 4.7.2), and various elements such as arguments and return value containing expressions. Details of expressions are described in Section 4.7.6. Their usage in the plan header is described here.

4.7.1.1 Arguments

Like functions in programming languages, plans can have arguments to pass values from the caller (the parent plan) to the invoked child plan. Each module implementing the computation of one argument value is connected to one assignment module. This module does nothing but store its input when it is triggered to do so. The trigger is activated whenever a plan is instantiated. This can happen more than once because aborted plans can be retried. The output of the assignment module is connected to all modules using the argument value. This is to ensure that the argument value remains constant during the lifetime of the plan, even if the basis for its computation changes.

4.7.1.2 Return values

In contrast to many programming languages, Asbru plans can have multiple return values. They are implemented similar to arguments. Of course, the input of the assignment module is a computation module defining the return value and the output of the assignment module is connected to modules implementing the plan body of the parent plan, which use this return value.

4.7.2 Plan body

The body of an Asbru plan contains one of the following. Each of the items is discussed in detail below.

single-step. One basic (atomic) plan step. This is one of the following:

- **plan-activation.** The activation (or call) of another plan including argument passing and return-value assignment.
- **subplans.** A set of plan body elements, performed in a particular ordering, e.g., sequential or parallel.
- **variable-assignment.** The assignment of a value given by an arbitrary expression to a variable.
- **parameter-assignment.** The same for a parameter.

set-context. Changing the value of a context variable.

ask. Requesting input for a parameter from the user.

list-manipulations. List manipulations are not within the scope of this thesis.

if-then-else. This plan step consists of a condition and two branches which again consist of one or more plan steps.

cyclical-plan. The cyclical plan provides complex and powerful means to describe repeated actions.

for-each-plan. The for-each plan is used on conjunction with lists, and thus not discussed in this thesis.

iterative-plan. The iterative plan performs a series of single step repeatedly. It is a simple alternative to the cyclical plan for list manipulations, and not discussed in this thesis.

refer-to. This element means that the plan body of another plan is copied here, it does not influence the execution of a plan.

user-performed. This plan is performed by the user.

In the following, the implementation of these elements is described ordered by importance for the understanding of their interaction. This means, that first the plan activation is discussed, followed by the subplans element. Both are key to the plan hierarchy and the interaction between plans and their superplans (or parent plans) and subplans (or child plans). Then the remaining elements are discussed in the order above.

4.7.2.1 Plan activation

The plan activation specifies:

- The name of the plan to be activated.
- A list of arguments for that plan.
- A list of assignments of return values of that plan to variables of the calling plan (an Asbru plan can have more than one return value).
- A time-annotation limiting the execution time of the called plan.
- The typical duration of the called plan, which does not influence plan execution and is therefore not discussed here.
- A single step to be performed if the called plan aborts. In most cases, this will be a plan activation or variable assignment.
- Another single step to be performed if the called plan is suspended.

Only the first item in the list is mandatory.

This is translated to abstraction modules as follows. If there are no arguments, return values, *on-abort* and *on-suspend* plans, then only one PlanStateModule is created[16]. In all other cases, a PlanActivationModule is created. It is connected to the plan containing the Asbru *plan-activation* as its child. It has the following subplans:

[16]The actual type of it depends on the type of the activated plan (sequential, parallel, etc.).

- For each argument, an AssignmentModule is created. If the value if an argument is simply taken form a variable or parameter, then the output of the corresponding module is connected to the input of the AssignmentModule. Otherwise, abstraction modules are created to represent the expression found in the *argument-value* element.

  The output of the AssignmentModule is connected to the PlanActivationModule since these modules represent subplans, and to a newly created PlanModifiedParameterModule which stores the value of the argument for later use in the activated plan.

- For each return value, another AssignmentModule is created, following the same scheme as above. The input of each of these modules is connected to a newly created PlanModifiedParameterModule which stores one return value each (Asbru plans can have multiple return values). The output of the AssignmentModule is connected to those PlanModifiedParameterModule to which the return values are copied (at the Asbru level). These modules had been created before when the parser found, the Asbru element *variable-def* or another definition.

- A module for the plan named in the plan-activation is created. It is connected to the PlanActivationModule as one of its children.

- If the *plan-activation* contains an *on-abort* clause, then another PlanStateModule is created and connected to the PlanActivationModule as one of its children. The type of this PlanStateModule depends on the type of the plan which is activated when the originally activated plan is aborted.

- The same happens for the *on-suspend* clause.

Details on parsing the time annotation are given in Section 4.7.5.

4.7.2.2 Subplans

The key element for building a plan hierarchy is *subplans*. It contains the following information:

Type of the plan. This is one of the following four values. For each, there is a plan module described in Section 4.6.3.

parallel. All children are instantiated and they become *considered* immediately. Once they are all ready to become *activated*, they enter this state at the same time. After this, there is no further synchronization between them.

sequential. Only the first child is instantiated. After it is either *completed* or *aborted*, the next child is instantiated an so on.

unordered. All children are instantiated immediately and there is no synchronization between them.

any-order. All children are instantiated. The first one which is ready to be activated is activated. The next one is activated if the first one is either suspended, completed or aborted. The suspended plan is not reactivated as long as the second one is activated.

A list of single steps. These are most often plan activations calling other plans, called children or child plans. They are connected to the above plan module as described in Section 4.6.2. If a single step is not a plan activation, then a suitable module is instantiated to form an anonymous child plan, e.g., to implement a variable assignment.

The continuation specification. This specifies which children must complete before the parent completes, either in the form of a logical expression or as a count. This is implemented by a tree where the nodes implement the logical combinations via AndModule and OrModule and the leaves are PlanStateExtractionModules combined with ComparisonModules. If a count is specified, the a CountModule is feed by an OrModule combining the ComparisonModules monitoring complete state of all child plans.

The propagation specification. This specifies those children, for which a failure leads to the failure of the parent. It is mapped to abstraction modules in a similar way as the continuation specification.

The flag *wait-for-optional-subplans*. If it is set, then parent plan waits for all children, not only those prescribed in the continuation specification. This is a popular way to specify *wait-for all*, but *abort-if* only for the plans given in the *continuation specification* fail.

In this case, an additional tree of abstraction modules is added to that resulting from the continuation condition (if there is one), which implements the condition that all child plans must be terminated, i.e., either completed or aborted.

The flag *retry-aborted-subplans*. If it is set, then subplans are put back to plan state *considered* once they reach plan state *aborted*.

This is implemented by providing variants of the four types of plan modules described above.

4.7.2.3 Variable assignment, parameter assignment, and set context

The assignment of a value given by an arbitrary expression is mapped to an AssignmentModule, which simply outputs the input at the moment at which it receives the signal to initialize as a child plan. It then completes as a plan. In the case of a complex expression, this expression is mapped to a tree of suitable modules, like AddModule.

4.7.2.4 Ask

This plan step requesting input for a parameter from the user. This is mapped to an AskModule described in Section 4.6.3.9.

4.7.2.5 If-then-else

This plan step consists of a condition and two branches which again consist of one or more plan steps. It is mapped to an IfThenElseModule as described in Section 4.6.3.8.

4.7.2.6 Cyclical plan

The cyclical plan provides complex and powerful means to describe repeated actions. It is mapped to an CyclicalPlanModule, which is described in Section 4.6.3.10

4.7.2.7 User-performed plan

This plan is performed by the user. A UserPerformedPlanModule is instantiated. The sole role of this module is to leave a trace in the log to represent the recommendation of the guideline model, but latter implementations can also realize complex user interaction in this case.

4.7.3 Conditions

All conditions contain the following parts.

Temporal pattern. This is the core of the condition, containing combinations of parameter propositions, but also other elements. It is described in Section 4.7.4.

Flag *overridable*. If set to yes, then the user can cause the plan state transition mediated by this condition, without the condition itself holding.

Flag *confirmation-required*. If set to yes, then the plan state transition does not take place before the user confirms it.

To implement the functionality of the two flags, a ConditionModule is inserted between the output of the module implementing the temporal pattern and the input of the plan module. Its sole function is sending requests for confirmation to the user interface and receiving confirmations and override commands.

In addition to the above, two of the conditions contain extras: First, the setup precondition contains a time annotation labeled *waiting-period*. It describes the time during which plans to achieve the state described by the setup precondition are started. It is treated like the time annotation in plan activations (compare Section 4.7.5.1).

Second, the abort condition permits the modeler to specify whether it is valid for activated or suspended state only, or for both. By default, it is valid for both states. The plan modules have two inputs, abort-from-suspended-condition and abort-from-activated-condition and they are connected according to this flag.

4.7.4 Temporal patterns

The Asbru element *temporal-pattern* is a placeholder for one of the items discussed in the following subsections.

4.7.4.1 parameter-proposition

This describes a value for a parameter and an interval during which this value must hold, as well as a context which must be given. The value is described in one of several ways.

value-description defines a comparison between the parameter and an expression. Depending on the comparison operator (less, less or equal, etc.), this is mapped to the corresponding comparison module (Section 4.2.3.6). The inputs to this module are the module generating the parameter value, and the root module of the tree implementing the expression. Mapping the expression is described in Section 4.7.6.

value-range defines upper and lower bounds for the parameter. This is mapped to an AndModule (Section 4.2.3.1) with two comparison modules as inputs. Depending on the Boolean flag *include-limit* in *upper-bound* resp. *lower-bound*, the comparison operator is *less* or *less-or-equal* resp. *greater-or-equal*. The bounds themselves are expressions handled as described in Section 4.7.6.

is-known-parameter evaluates to true if this parameter is not *undefined*. This is mapped to the DefinedModule (Section 4.2.4.3).

is-not-known-parameter is the complement of the above. It is mapped to a NotModule fed by a DefinedModule.

The context is given as one of the following.

context-ref gives the name of a context variable of type Boolean. It is simply mapped to the module which supplies the value of this context variable.

context-combination combines two context references using Boolean operators in a recursive way. It is mapped to a logical comparison module corresponding to the given operator. The inputs to this module are the modules implementing the arguments of the Asbru element.

context-not is the negation of the given context value. It is mapped to the NotModule.

one-of specifies a list of constants and a context variable. If the context variable matches one of the given constants, then the context is given. It is mapped to an OrModule fed by EqualModules each of which compares the referenced context variable with one constant.

any is the default value specifying the context which is always given. It is mapped to a ConstantModule for the value *true*.

The interval during the which both the value and the context must meet the prescribed criteria is defined by a *time-annotation*. Particularities of parsing this element are discussed in Section 4.7.5.

The output of the modules monitoring the value and the output of the modules monitoring the context are both fed into an AndModule. The output from this module is the input of a parameter module. Depending on the reference point of the time annotation, parameter module for fixed reference point (compare Section 4.5.2.1), repeated reference point (Section 4.5.2.2) or for reference point *now* (Section 4.5.2.3) is used.

4.7.4.2 timeless-parameter-query

This is either *is-automatic* or *is-manual* and checks whether a parameter is entered automatically or manually. There is no temporal abstraction involved. This is evaluated by the Asbru compiler based on the data in the configuration file of the project which wraps the protocol model at hand. Accordingly, a ConstantModule is generated outputting true or false, depending on the data in the configuration file.

4.7.4.3 plan-state-constraint

This describes a plan state for a plan and an interval during which the plan must be (or must have been) in that state. It resembles the parameter proposition. Instead of connecting to a module providing the parameter value, a plan module is connected to an EqualModule fed by a PlanStateExtractionModule (Section 4.5.3) and a ConstantModule representing the plan state.

4.7.4.4 temporal-constraint

This defines the qualitative temporal relation of intervals which again are defined by temporal patterns. Possible values of the relational operator are: before, overlaps, starts, equals, meets, during, and finishes. It is mapped to one of the modules described in Section 4.5.4

4.7.4.5 constraint-not

This defines the Boolean negation of a temporal pattern. It is mapped to a TemporalPatternNot module (Section 4.5.5.1).

4.7.4.6 constraint-combination

This defines a Boolean combination such as a conjunction or disjunction for temporal patterns. It is mapped to the modules described in Section 4.5.5.1 to 4.5.5.3.

4.7.4.7 count-constraint

This contains a temporal pattern and a minimum number of occurrences for this temporal pattern. Only if this number is reached, the count constraint is fulfilled. This is implemented by a CountModule (Section 4.5.7.3) and an EqualModule (Section 4.2.3.6). The module implementing the temporal pattern feeds the CountModule which again feeds the EqualModule.

4.7.4.8 simple-condition

This defines a relation between two instantaneous values. These values can be parameters or variables. In both cases, an AssignmentModule is used to forward the current value at the time of evaluation to the comparison module which implements the comparison operator. Of course, constants can be used in the comparison, too, in which case a suitable constant module is connected to the comparison module.

4.7.4.9 refer-to

This element points to another condition containing the definition of a temporal pattern. It is a means of syntactical reuse. Consequently, the output of the modules implementing the referenced temporal pattern is reused, too.

4.7.4.10 none

This element verbosely states that there is no condition. It is used in conjunction with the *confirmation-required* flag where a plan is started at the user's discretion. Like missing conditions, conditions defined as *none* are mapped to a constant element generating a *fulfilled* episode data-point.

4.7.4.11 to-be-defined

This shows that this condition needs to be defined but is not yet (in the design process of a guideline). Execution of an Asbru plan library containing this element is refused by the Asbru compiler.

4.7.5 Time annotations

Time annotations in Asbru are given by Earliest and Latest Starting *Shift* (ESS and LSS), Earliest and Latest Finishing *Shift* (EFS and LFS), Minimum and Maximum Duration (MinDu and MaxDu), and the Reference Point (RP) to which the shifts refer to. All of these seven values can be arbitrary expressions. The reference point can also contain two special placeholders: *now* and *self*.

There are two different modes of using time annotations as far as plan execution is concerned:

1. Time annotations in *defaults* in *plan* and *plan-schema* in *plan-activation* govern plan activations.

2. Time annotations in *parameter-proposition* and *plan-state-constraint*, govern the matching of measured values, either in conditions or in definitions of temporal data abstraction.

4.7.5.1 Restricting plan activation

Time annotations which restrict the time of plan activation are mapped to alarms which are processed by the PlanModule implementing the plan in question. Setting the alarms is performed by the PlanModule. Computing their values (time points) is performed by a tree of abstraction modules implementing the expressions in the shifts in the time annotation and the addition of the reference point in the first four cases, as detailed below.

In the Reference Point (RP), special value *now* is mapped to the time at which the plan is initiated, i.e., the time at which the plan activation is evaluated. Special value *self* is mapped to the time of the first activation of the parent plan. This is implemented by inserting an AssignmentModule fed by a NowModule as the first child of the plan.

Earliest Starting Time. This is the absolute time point computed by adding ESS and RP. Changing to *activated* state for the first time is delayed until this alarm arrives.

Latest Starting Time. This is the sum of LSS and RP. If the plan is still in *considered* or *possible* state, it is rejected.

Earliest Finishing Time. This is the sum of EFS and RP. If the complete condition and continuation condition are fulfilled, the plan still does now complete before this alarm arrived.

Latest Finishing Time. This is the sum of LFS and RP. If the plan is not completed or aborted when this time arrives, then the plan is aborted.

Minimum Duration. This is the value as defined in the time annotation. It is added to the first time of plan activation internally by the PlanModule. The result is treated like the Earliest Finishing Time.

Maximum Duration. This is also treated like above. The resulting alarm is treated like that for the Latest Finishing Time.

4.7.5.2 Defining the monitoring process

Time annotations in *parameter-proposition* and *plan-state-constraint* map to arguments to the monitoring modules described in Sections 4.5.2.1 through 4.5.2.3.

Special value *self* is mapped as above. However, it is not permitted in parameter definitions, because they are not part of any plan. The same holds for filter and setup preconditions, because the plan is not activated then, and these conditions cannot use an unknown time point in the future as a reference point.

On a logical level, values in the condition are calculated once when the time annotation is (first) evaluated. These are the following time points depending on the usage of the time annotation.

- Time annotations in filter precondition: When the plan is initiated.
- Time annotations in setup precondition: When the plan becomes *possible*.
- Time annotations in suspend, complete, and abort condition: When the plan (first) is activated.
- Time annotations in reactivate condition: When the plan is (first) suspended.
- Time annotations in parameter definitions: At program start.

There is no need to insert AssignmentModules in this case since the modules monitoring parameter propositions read these inputs only once, when initialized.

Parameter changes as reference point are mapped to a repeated reference point. It is most useful in parameter definitions. In conditions it can be used, here the condition is fulfilled if an episode is found for anyone of the reference points. For details see Section 4.5.2.2.

4.7.6 Expressions

Expressions can be used in many places. Only in few cases, the location of the expression influences its evaluation.

They are always mapped to a tree of abstraction modules.

Constants are mapped to ConstantModule.

Variables, arguments and parameters are mapped the modules producing the referenced value. I.e., no new module is instantiated during parsing this part of the expression, only a new connection to the output of an existing module is created.

Arithmetic operations are mapped to the corresponding arithmetic modules like AddModule, described in Section 4.2.1.1.

Time point *now* is mapped to the NowModule, which always outputs the current valid time.

Time point *self* refers to the first activation of the plan containing the expression. It is mapped to the helper module SelfModule which takes the PlanStateDataPoints output by the PlanModule which implements the current plan, and outputs the valid time of each such data point which marks a change from *possible* state to *activated* state.

Plan state transitions are mapped to a ValidTimeModule fed by a ComparisonModule fed by a PlanStateExtractionModule fed by the current plan. The second input to the ComparisonModule is a ConstantModule representing the given plan state.

4.7.7 Definitions

The parts of a domain definition which are relevant to data abstraction are:

- Type definitions: definition of qualitative and numerical scales.
- Parameter definitions.
- Context definitions.
- Constant definition.

4.7.7.1 Type definitions

qualitative-scale-def describes a qualitative scale or data type. On the Asbru level, the values of a qualitative parameter or variable are referred to by their symbolic names such as “low” or “medium”. In the data abstraction process, each of them is represented by an integer, starting with zero for the first entry in the list of qualitative values. Therefore, the operations minimum, maximum are defined for qualitative values as well as comparisons such as “less than”.

Qualitative scales can have secondary entries, which represent a group of primary entries. E.g., the secondary entry *low* can stand for the primary entries *very low* and *moderately low*. Referring to *low* is the same as referring to "very low or moderately low" and is implemented exactly in this way. I.e., the plan execution unit produces the appropriate disjunction.

For such secondary entries, the Asbru compiler must add an OrModule and two or more EqualModules where otherwise it would just use one EqualModule.

numerical-scale-def allows the user to add scales to the built in ones for numerical values. The plan execution unit handles various conversions between compatible scales, e.g., the user can compare a variable containing "1 km" to a parameter delivering values in "cm". The data abstraction process handles scales date and time as integers, and all other scales as float. Each scale has a default unit and values coming from the data abstraction unit are interpreted as having this is the unit. In summary, while the plan execution unit handles complex modes of accessing values, the data abstraction unit is relieved from this and only deals with simple numbers.

4.7.7.2 Parameter definitions

parameter-ref states that the input to this parameter (i.e., abstraction step) comes from another parameter. Consequently, the output of another module is used as an input to the module implementing the enclosing statement of the *parameter-ref.*

raw-data-def defines one original input from outside the system – either user input, file input, or data received online from measuring devices.

The attribute *mode* specifies whether input is *manual*, *automatic*, or *automatic-or-manual*. In addition, this can be specified in the project file for this plan library. This information is not used for defining the source, but only in Asbru queries at runtime.

For file input and online input, the attribute *channel-name* specifies the column containing the data for this parameter. For interactive input, attribute *user-text* supplies a message to display when the user is asked for the value.

If attribute *use-as-context* is *yes*, the values of this parameter are used as *context variables* independent from other usage. Section 4.7.7.4 gives further detail.

If minimum or maximum are defined for value or increase, then a limit check module is instantiated and the data source is connected to this module. Any references to this parameter are connected to the output of the limit check module. Otherwise a *raw-data-def* does not lead to the instantiation of any abstraction module, but to an entry in the reference table mapping this parameter to the source given in *channel-name* and the selection of the input source by the user.

calculation-def defines one of a set of different operations depending on the value of attribute *operator*. For each value of this attribute there is an abstraction module the name of which is the value of the attribute with "Module" appended, e.g., AddModule for *operator* = *add*.

logical-combination-def defines one of four logical combinations of parameters, depending on the value of attribute *operator*: and, or, xor, not. For each case there is an abstraction module with that name, appending “Module”.

spread-def defines a Spread as described in Section 4.4.4. Attribute *type* specifies whether the Spread is based on the standard error or the quantile.

slope-def represents the slope or ascent of a regression line. It is mapped to a SlopeModule.

standard-deviation-def represents the standard deviation in the calculation of the regression line and is mapped to a StandardDeviationModule.

end-point-def represents the end point of the regression line which is a rather lively predictor of further measurements. It is mapped to a EndPointModule.

time-to-alarm-def defines the time from the end of the regression line to the intersection of the regression line with the limit given as an argument. Compare section 4.3.4.2.

change-def represents the difference between a previous measurement and the most recent one. The temporal distance between the two is given in *interval* which is a constant expression. This element is mapped to a ChangeModule.

average-def defines the average of a parameter for the given time window. It is mapped to a TimeWindowModule connected to an AverageModule.

qualitative-parameter-def defines qualitative values based on either numerical input or a Spread. In the first case the numerical data is directly feed into a RawDataBasedQualitativeModule. In the second case, the numerical input is feed to a LinearRegressionModule connected to a SpreadModule which again is connected to a SpreadBasedQualitativeModule.

In each case context sensitive selection of the limits of the qualitative regions is performed by a separate group of abstraction modules as described in Section 4.7.7.4.

logical-dependency-def implements alternatives (a.k.a. if-then-else) in the data abstraction process. It consists of a series of pairs each formed by a condition and a result which is output if the condition holds.

To implement this statement, the temporal pattern forming each of the conditions is implemented as described in Section 4.7.4. The result of the first one, representing the first condition, is feed as first input into a SwitchModule (described in Section 4.2.3.5). The second is used as third input, the third as fifth, and so on. The even-numbered inputs of the SwitchModule are either obtained directly from the sources or from ConstantModules, depending on the formulation of the statement in the plan library.

After the last pair of condition and value reference, the element *default* can give the value which is used whenever none of the conditions holds. This is mapped an additional input to the SwitchModule. If no default is given, then this additional input is omitted.

boolean-def defines the outcome of monitoring a parameter proposition as a Boolean value. It is implemented by instantiating

1. several ConstantModules and modules comparing them with parameters to implement monitoring the context of the parameter proposition;
2. other ConstantModules and modules comparing them the parameter of the parameter proposition to implement monitoring the value description;
3. an AndModule combining the output of monitoring item 1 and 2;
4. a ParameterPropositionModule receiving the time annotation as argument and the output of item 3 as input;
5. a ParameterPropositionToBooleanModule transforming the output of item 4 to a Boolean.

4.7.7.3 Context definition

Context variables come in different forms: Internally set context, directly entered context, and abstracted context. In the first case, context variables are declared using Asbru element *context-def* and set by plans using *set-context*. This is mapped to an *AssignmentModule*. In the second case, the value of the context variables is entered by the user in parameters defined by *raw-data-def* with the attribute *use-as-context* set. In the third case, the value of a context variable is defined by a parameter abstracted from others. Again, attribute *use-as-context* is set in the definition.

In cases two and three, the abstraction graph does not differ because of the attribute *use-as-context*. This attribute only influences the name matching performed by the Asbru compiler.

4.7.7.4 Context usage

There are three different issues in Asbru in which the context plays an important role: Mapping quantitative values to qualitative ones, monitoring parameter propositions, and reasoning about the effect of parameter dependencies and plan execution. The latter is not part of the work described here, for reasons given in Section 5.4.4.

Context-dependent assignment of qualitative values. A quantitative parameter is abstracted to a qualitative one using a list of limits. If the numeric value of the quantitative parameter lies between the first two limits, it is mapped to the first qualitative value. If it lies between the second and the third limit, it is mapped to the second qualitative value.

In Asbru, there is not a single fixed set of limits for such a mapping. Instead, several sets of limits can be given, each for a different context. A context is defined by a Boolean combination of pairs of context variables and their values, e.g., preterm child is true and mode of ventilation is CPAP.

In the abstraction process this is implemented by

1. one or more modules for each context, outputting true if this context is given;
2. several groups of ConstantModules producing the limits for every context;
3. a SwitchModule selecting that set of limits which is suitable for the given context – the output of the first step are the odd-numbered inputs, those of the second step the even-numbered ones, if the last context in the limit declaration is *any*, then those limits are the default in the SwitchModule;
4. a RawDataBasedQualitativeModule mapping the quantitative input to qualitative values based on the selected limits.

Context of parameter propositions. Parameter propositions contain a value description for a certain parameter, a time annotation, and a context. They are fulfilled if the parameter's value meets the description for the time specified in the given context. In practice this means that the context must be given and the value must fit the description in the prescribed time, i.e., the context description and the value description are simply *and*-connected.

This is implemented by feeding the output of the abstraction modules implementing the value description and of those implementing the context into a AndModule. The output of the latter is the input which is monitored by the ParameterPropositionModule.

4.8 Discussion

In the above sections of this chapter, I presented my solutions to the objectives, which include novel algorithms and a framework tailored to the challenge. In this section, I wrap up the reasoning behind the designs, before proceeding to the evaluation in the next chapter.

Uniform framework

Implementing complex abstractions by decomposing them into small parts which can be connected freely in a network is a standard technique in system design. There are three possible approaches to implementing a network of abstraction modules: push, pull, and mixed. In a push network, new data pushes the abstraction process – whenever a new datum arrives, all the abstractions that are based on it are calculated. In a pull network, abstractions are only calculated on demand whenever their result is required by a further processing step.

Push networks are advantageous where each bit of input information must be processed without delay. Pull networks are advantageous where access to the input data is infrequent, some of the abstractions of the input would never be used and delays in responding to input are not significant.

In a push network of i inputs, on average a abstractions of each input, and a depth of d, ia^d abstractions are generated – and they must be stored until used. In a pull network, intermediate abstractions are not stored, which cuts the storage effort but introduced the danger that intermediate abstractions are calculated multiple times if used by different other abstractions.

Mixed mode networks combine push and pull networks to circumvent these problems. The idea is to generate those intermediate abstractions which are expected to be reused and store them for later use. For example, the IDAN framework [19, 20, 10] developed significant sophistication storing and retrieving the intermediate results on various layers of abstraction in a database and in memory. To achieve quick response times in a high-frequency setting, I avoided the use of databases. After exploring mixed mode, I opted for a push architecture because the simpler principal architecture permitted the implementation of other features (described below) to meet the particular demands of complex networks of high-frequency abstractions.

My key idea for reducing the data volume in a push network is to reduce the frequency at which an abstraction produces new output as soon as possible in the abstraction process. This is a two-part process. On the one hand, the implementation of the abstraction algorithms must prevent the generation of redundant output, and the overall framework must handle data streams of different and varying frequency. On the other hand, the knowledge engineer must configure the abstractions in such a way that the frequency of calculation is reduced before the data is fed into a complex network containing a large number of abstractions. Practical experience (compare Section 5.1.1) shows that such a setting is compatible with the practical reasoning in medicine – first, potentially noisy, high-frequency data is transformed into qualitative values, for which it is desired that they are steady. This steadiness translates to low data volume if only new values are forwarded internally. My design of the abstraction framework ensures this and only invokes abstraction steps if one of their inputs changed.

Online-algorithms for monitoring temporal patterns

In an analogy to the general framework, monitoring the patient in the process of executing Asbru plans which encode the treatment could be implemented in a mixed mode approach. Such an approach was envisioned in our early work [93]. However, this design would require complex data structures to keep track of temporal information describing those abstractions which are of current importance. More importantly, the complex semantics of Asbru would make render the handling of these data structures computationally intensive and error-prone.

I therefore changed the design to a push scheme. This required the design of complex algorithms described in Section 4.5.2. Besides testing them against the requirements set forth by the Asbru semantics, which is not described in this thesis, I describe the upper limits of the computational effort associated with each of the processing steps in Section 5.2.5.

Previous work in the field [97] used pull approaches, implementing the monitoring as a set of database queries to be performed at regular intervals. For high-frequency domains, the data volume and the short processing intervals are preclusive to such an approach.

Integration of plan execution

The semantics of Asbru are defined with a pull approach in mind. Again, I had to design a set of algorithms to implement these semantics within a push network. The benefit of this effort is the immediate response to any change in an input value, propagated to all plans. In high-frequency domains, this is a significant advantage over the repeated query of conditions foreseen in the original design of Asbru 6.4 [95].

When a condition is fulfilled, this results in a plan state change, which may influence the plan state of child or parent plan of the plan concerned. This state change is communicated to other plans as a abstract data point much like an abstraction of an input data point. Section 5.2.1 shows that upper limit of the response time to plan state changes is proportional to the product of the depth of the plan hierarchy and the number of backlinking modules, i.e., those modules where a plan state change occurs which is relevant for another plan.

Sliding time windows for statistical analysis

To create reliable abstractions of high-frequency input data, I combined descriptive statistics with the concept of a focus of attention which lies on a time window comprising the most recent past. In knowledge acquisition meetings with physicians, I found that they repeatedly refer to features in the input data occurring in the most recent 1 or 3 or 5 minutes.

An important feature in this context is the possibility to set the frequency at which the abstraction is performed independently of the size of the time window. This allows for overlapping time windows, e.g., evaluating the most recent 3 minutes every minute, which uncouples response time from window size. It also reduced the chance that a relevant temporal pattern is not found because it is split by the time window limit.

Coping with noisy data

Building on existing work on knowledge-based error detection and removal, I combined the rule-based approach with the statistical approach applied to a sliding time window. Summarising the data for regular intervals using average or median is not a new method. Also deriving trends from the most recent measurements has been included in previous research [29, 67, 92]. However, I am not aware of an approach combining these techniques with guideline execution for a high-frequency setting.

The Spread algorithm is a novel contribution to the set of time-window based abstractions. I designed it to model the physicians' perception of "the majority of the measurements" in a certain time window. The algorithm has a set of variations to optimally meet the requirements of the application to develop, e.g., the outliers to be ignored can be defined based on standard deviation, standard error, or quantiles. These three measures alternatively define the width of the Spread. A separate limit on this width excludes episodes in which the input signal oscillates too much. This suspends the abstraction process for times on unreliable input. If desired, explicit reaction to this situation can be defined by making *undefined* output from the Spread algorithm the *filter-precondition* (compare Section 4.6.2) of the plan implementing the reaction.

A further variant refers to the modality of mapping the Spread to qualitative values. With the *memory-on* flag set, the algorithm reacts very conservative to changes in the input. It only changes the output value if the whole width of the Spread entered a qualitative region different from the previous one. This serves as a second layer of oscillation suppression. First, oscillations within the time window only increase the width of the Spread but do not lead to oscillations of the Spread itself, only changes in the average value in a time window do. Second, oscillations of the Spread do not result in oscillation of the qualitative output as long as the variation is less then the Spread width.

With the *memory-on* flag not set, the qualitative value is found based on a *normal region*, which is the range of values for which no actions need to be taken. Without consideration of the past, that margin of the Spread is considered which is nearer to the normal region. The qualitative region within which this margin lies defines the output value. This mode is appropriate if there is a normal region for which corrective actions are not required, and if the second level of smoothing is not desired.

Utility functions

Providing a set of elementary functions performing arithmetics and logical combinations, as well as supplying system information such as the day of the week, is a necessary precondition for the combination of the more sophisticated parts. It is not innovative in itself, but greatly extends the usability of a toolbox in practical applications.

Bridge to Asbru

Mapping the language elements of Asbru to the modules described in the previous sections is a precondition to applying the modules to execute Asbru plans. Demonstrating the large extent, to which Asbru can be mapped to solutions presented in this thesis, Section 4.7 forms a bridge to the next chapter, which describes the evaluation.

Chapter 5

Evaluation

This chapter groups the evaluation of the solutions presented above into two parts: practical evaluation in Section 5.1 and theoretical discussion in Sections 5.2 through 5.4.

Practical evaluation focussed on two parts, the Spread algorithm and the plan execution. In the field of medicine, practical evaluation is a major undertaking. Quality assurance standards in medical care prescribe that the correctness and usefulness of new systems to be tried out must be proven and acknowledged by ethics commissions; patients (or their parents) must consent in advance before; and further development of the technical solution is impeded by the need to resubmit the proposal each time the changes are of any significance.

At the same time, there are many organisational reasons for under-use or non-use of a new system. And even if the system is used, it is very difficult to deliver a sound proof of a clinical impact [49].

However, we completed a controlled clinical trial in very close cooperation with medical partners, with very favourable results, as described in Section 5.1.1.

Execution a clinical protocol in daily practice is pursued within the OncoCure project described in Section 5.1.2.2. However, in this case, tangible clinical results are still well ahead. Two other projects used the Asbru interpreter. The focus of the Protocure project (Section 5.1.2.1 was a more theoretical one, and that of the Remine project (Section 5.1.2.3) was on data integration.

In all three projects above, low-frequency domains were modelled. We were not able to influence the decisions in favour of high-frequency domains. An important reason is that there are very few guidelines concerning treatment at intensive care units and a host of guidelines for primary care, cancer treatment, and other low-frequency domains.

The lack of efficient temporal data abstraction at the point of care makes it unattractive to formulate protocols or guidelines for high-frequency domains; while in low-frequency domains, they can readily be applied, relying on simple concepts which are already available in the electronic patient record.

To complement the practical evaluation of those parts where we could bridge practice to my ideas, I discuss the complexity of the algorithms (in Section 5.2), how the proposed solutions meet the objectives (in Section 5.3), and their limitations (in Section 5.4).

5.1 Practical Evaluation

The work described in this thesis has been deployed by two lines of practical application. First, the Spread algorithm described in Section 4.4.4 forms the basis of a system to control the supply of oxygen in neonatal intensive care. It is described in Section 5.1.1. Second, the framework of temporal data abstraction and guideline execution described in Section 4 was implemented. It is since used in a series of research projects, described in Section 5.1.2.

5.1.1 Data abstraction for artificial ventilation

Artificial ventilation of neonates in an intensive care unit is difficult, because of the measurements delivered by the pulsoximetry device oscillate to varying degrees. It is time consuming because the patient state, and thus the oxygen demand, changes continually which would suggest frequent evaluation of the patient state with corresponding adjustments of the level of oxygen supply. And it is important to optimize it, because inadequate oxygen supply is associated with damage of the retina.

5.1.1.1 Initial evaluation of the Spread algorithm

Based on the comparison of physician perception of the oscillating graph delivered by the pulsoximetry device, I introduced the original Spread algorithm (compare Section 4.4.4).

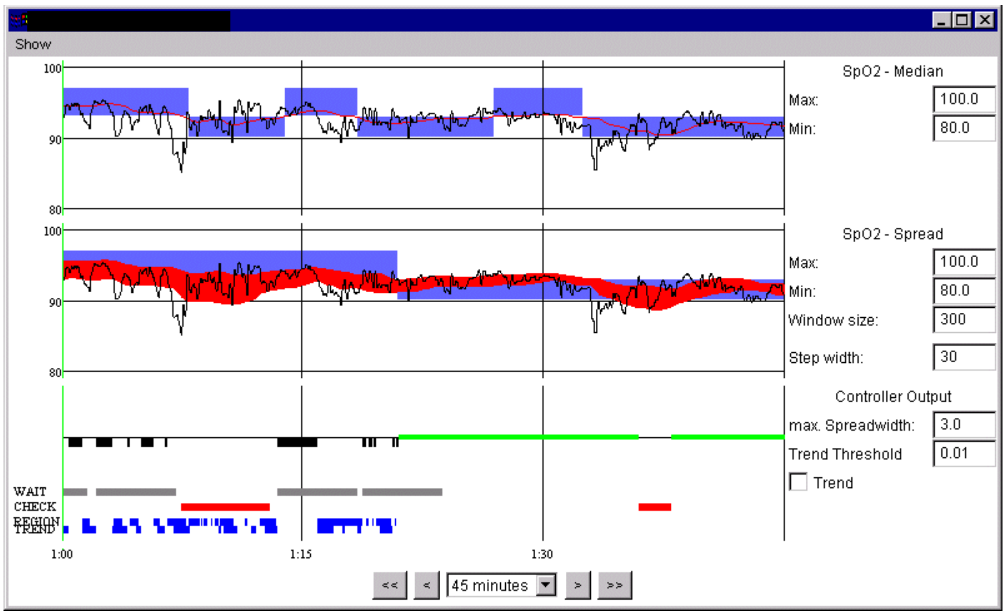
Evaluation setting. We compared the Spread algorithm to median filtering, which is the established and simpler method of smoothing undesired oscillations, in this field of application. The retrospective evaluation comprised recordings obtained from 10 patients [128].

Evaluation results. Figure 5.1 shows the data visualisation used throughout the evaluation process. The top graph shows the measurements in black, the median as a red line, and the qualitative values associated with the median, as blue boxes. The graph in the middle shows the same measurements, but with the Spread in red, and the qualitative values derived from Spread in blue. As can be seen, the Spread suppressed transient changes in the qualitative output which the physicians would rate as irrelevant.

"In a total of 126 hours (median 12, range 324) the adjustment based on the Spread changed 148 times compared to 519 changes in the adjustment based on the median" [128, p. 225].

5.1.1.2 Addition of complex knowledge-based abstractions

Although "we were able to show that the Spread leads to a more stable judgment of the patients situation and therefore reduces the number of unnecessary adjustments of FiO_2" [128, p. 225], discussing the records in detail with physicians revealed a set of situations, in which the Spread algorithm, if used in isolation, is not sufficient.



SpO2 - Median
Max: 100.0
Min: 80.0

SpO2 - Spread
Max: 100.0
Min: 80.0
Window size: 300
Step width: 30

Controller Output
max. Spreadwidth: 3.0
Trend Threshold 0.01
☐ Trend

Figure 5.1: Visualisation of abstraction methods for artificial ventilation.
Top row: raw data in black, the median as a red line, and the qualitative values associated with the median, as blue boxes.
In the middle: the same raw data, but with the Spread in red, and the qualitative values derived from Spread in blue.
At the bottom: Wait mode shown as grey bars, check mode in red, and postpone mode in blue. Black bars above the grey ones indicate the internal resolution to decrease the oxygen supply, which is overruled by the other modes. Long green bars indicate periods of optimal oxygen supply [128].

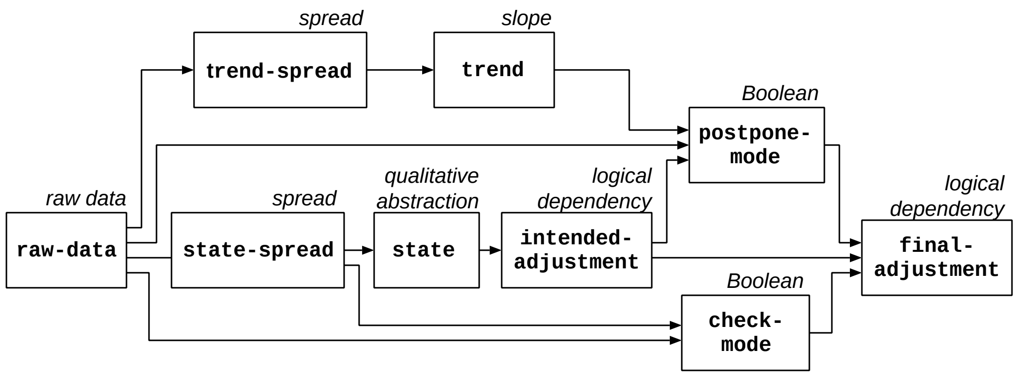


Figure 5.2: Abstraction graph for artificial ventilation control. Raw data from the pulsoximeter device flows from left to right via a set of abstraction modules configured according to domain experts' suggestions [128].

As a consequence, we devised the abstraction graph shown in Figure 5.2, which I implemented. In addition to the Spread evaluating the patient state over a longer period, a second Spread is calculated for the most recent minute. In the final design, this Spread only contributed the trend for this minute.

We introduced four modes of operation for our ventilation controller [128].

Postpone mode is entered when the recent trend, or the very last measurement, leads in the same direction as the recommendation which would otherwise be issued by the controller.

Check mode is entered for a number of reasons each of which suggests that remaining passive is the best option in situation at hand. These reasons range from a set of validity checks on the input to the user putting the controller into suspend mode in order not to disturb a care routine which must lead to measurements which would confuse the controller (e.g., by taking off the sensor).

Wait mode is entered for a preset time period after a change in the oxygen setting was enacted. A major design principle of our system is to make changes gradually and observe their result before proposing another change.

Active suggestion mode is entered if no other mode is active, and if the oxygen saturation of the patient is not in the target region. In this mode, the recommendation based on the state Spread is shown on a laptop.

The lower region of Figure 5.2 shows states of the controller. Wait mode is shown as grey bars, check mode in red, and postpone mode in blue. On the black horizontal line above these bars we see black bars indicating the internal inclination to decrease the oxygen supply, which is overruled by the other modes. The long green bars indicate regions in which no other (overruling) mode is active, but the patient's oxygen saturation is in the target region which means that no change in oxygen supply is needed.

5.1.1.3 Open-loop evaluation

Evaluation setting. We validated the above design by controlling the oxygen in an open-loop setting, in which a member of the clinical staff performed the changes to the oxygen supply according to the suggestions generated by our temporal abstraction system, running on a laptop connected to the pulsoximeter.

In a randomised crossover trial, 12 patients were randomized in 3 groups. Each group went through the different modes of treatment in another order. These modes were two baseline periods (always first and last), open-loop control by the controller, optimal manual control by an expert dedicated exclusively to this task and routine manual control by the normal care staff. Each of the five periods lasted 90 minutes.

Evaluation results. The target value of the study is the time spent in the target region of oxygen situation, i.e., the time during which the oxygen supply is considered optimal. It is denoted TT for *target time* in the following. In periods of optimal manual control, TT was 6.1% higher than for routine manual control. For our controller, it was 5.4% higher than for routine manual control [169].

5.1.1.4 Closed-loop evaluation

Based on a careful evaluation of the open-loop phase of the clinical evaluation, we introduced the quantile-based Spread and performed further fine-tuning regarding the size of the time windows used.

Evaluation setting. The resulting system was evaluated in a similar study design as before, as far as patient number and treatment modes are concerned. However, this time the laptop at the bedside directly controlled the oxygen supply via a serial connection.

Evaluation results. Again, our system performed equally well as the dedicated expert and both performed significantly better than the staff in the realistic setting in which they had to perform the full range of duties besides monitoring the oxygen supply. TT increased by 11% compared to routine care.

The staff was able to manually change the oxygen saturation in parallel to the controller. None of the changes by the controller was taken back by the study supervisor. The amount of oxygen adjustments by staff during these phases was only 11% compare to routine care. This indicates significant reduction in work load in addition to improved care [169].

The paper describing the study won the scientific award of the Society of Neonatology and Pediatric Intensive Care (Gesellschaft für Neonatologie und Prädiatrische Intensivmedizin) in 2005.

5.1.1.5 Multi-center study

Evaluation setting. A multi-center study at four German hospitals, started in 2008, currently explores the clinical benefit of longer blocks of continuous treatment using our controller. Following a run-in phase of three hours, the ventilation is controlled by our controller for 24 hours followed by 24 hours of routine control of ventilation by the personal normally in charge of this task. The order of the two 24-hour blocks is swapped for 50 % of the patients based on randomisation.

The aim is to demonstrate the benefit in patient state from our controller in a bigger picture. The definitions of optimal oxygen supply vary significantly between the three different hospitals which puts our controller to a test of versatility.

Evaluation results. First reports are promising. As of this writing, only one of the study sites completed the measurements. There, the performance of our controller was very satisfactory. However, an in-depth analysis of the complete measurements from all sites is necessary before publishing detailed results.

5.1.2 Plan execution

With my guidance, Peter Votruba, Michael Paesold and Gilbert Wondracek implemented an interpreter for a subset of Asbru, called Asbru Light, which implements the ideas presented in this thesis. It was originally developed as part of the Protocure project, and later used by the projects OncoCure and Remine.

The following describes these projects, the application of the interpreter within them, and the insight we gained in using it.

5.1.2.1 Protocure

The EU project Protocure[1], run as a two-phase FETopen project from 2002 to 2006. Its aim was to apply formal methods to improve guideline quality. Within this context, we modelled the US guideline for Jaundice in neonates and the Dutch Breast Cancer guideline in Asbru.

The choice of this guideline was not lucky for the purpose of this thesis, since breast cancer is a low-frequency domain, but it was lucky in terms of modelling, since we had the chance to closely cooperate with the Dutch organisation for guideline development, CBO.

The Asbru model was further translated to temporal logics and verified, but also executed by the Asbru interpreter [159].

The aim was to validate the model against a set of test data designed by physicians. Accordingly, the interpreter was designed for batch operation only, reading the complete patient data and outputting the recommendation in the form of the log file.

Our tests showed not only that the design could be implemented and that it covers the functionality demanded by the modelled guideline, but also that the implementation in Java was fast enough to run at an input data frequency of more than 1000 Hz on an standard notebook in 2006 [103].

5.1.2.2 OncoCure

The aim of the binational OncoCure project was to support the clinical care practice at the Medical Oncology Unit of the S. Chiara Hospital of Trento. The application domain was breast cancer again, but here with a strict focus on medical treatment. The

[1]`www.protocure.org`, last accessed May 9th, 2011

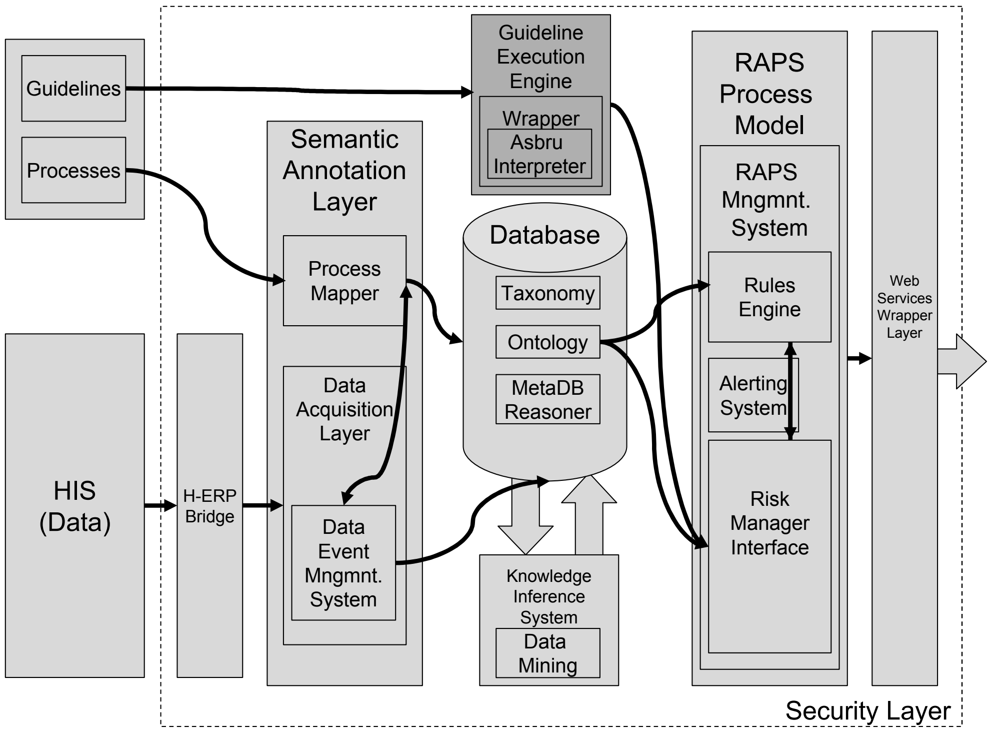


Figure 5.3: Remine system architecture.

basis of this Asbru model (which is unrelated to the one used in the Protocure project) was the protocol used at the clinic [47].

The interactive context required the creation of a set of wrappers which call the interpreter repeatedly, feeding it patient data, and translating the results to present it integrated into the legacy patient management system.

Feedback from clinicians on this system was positive. A detailed analysis of the impact of the system on the quality of care has not been conducted as of this writing, because of the optional use of the decision support system at this site.

The practical integration of the interpreter with the clinical routine at the clinic showed that the demand in a highly interactive setting poses new problems, beyond the formal integration of user confirmation into the protocol execution process. In particular, user preferences regarding the sequence of presentation of groups of options and series of questions impose restrictions on the modelling. It could also lead to the introduction of new language elements. While this has a strong modelling impact, it will also influence the further development of the interpreter.

5.1.2.3 Remine

The EU project Remine[2] seeked to reduce Risks Against Patient Safety by integrating various means to improve the quality of care into a complex framework. Within this

[2]`www.remine-project.eu`, last accessed Oct 20th, 2010

framework, the Asbru interpreter was used both for decision support at the point of care, and the analysis of what-if scenarios by a risk manager [124].

Figure 5.3 shows the Remine system architecture. It gives a clear picture of the limited importance of guideline execution in this project. Unfortunately, the untimely withdrawal of the consortium member in charge of the interpretation of the Asbru interpreter prevented the planned evaluation in clinical practice.

In the course of the project, four protocols were modelled, again demonstrating the appropriateness of Asbru for the task. The domains of application were: management of active low-risk labour, management of acute ischemic stroke, management of infection with Methicillin-resistant Staphylococcus Aureus, and management of medical treatment in the acute care for the elderly.

5.2 Complexity Analysis

This section discusses computational complexity and required storage space for all modules described in Section 4, and for the system formed by these modules, as a whole.

Since the focus in this thesis is on high-frequency applications, the focus of this section is on the effort per new measurement. Measured values are assumed to arrive in groups, at regular intervals, e.g., once per second, or millisecond. Such a synchronised time point of data input is called a macro time step, or time step.

Of course, not all input channels will deliver new input every macro time step. In practice, this makes an important factor, although the lower bound of time saving from this difference depends on the model, i.e., the Asbru plan library at hand.

In contrast to macro time steps, at micro time steps only internally generated data needs to be processed. This is only the case at circular links in the abstraction graph. Section 5.2.1 goes into details in this regard.

In this section, $O(n)$ stands for the upper limit of $f(n) = cn + x$ for any c and x given sufficiently large n. $O(1)$ stands for constant effort $f(n) = x$ not related to any changing factor.

5.2.1 The overall system

The complexity of the overall system is the sum of its parts plus the management overhead. There are three cases to consider for the overall computation taking place per macro step: Standard operation of abstraction modules, internal playback mode, and the communication loop between parent and child plan.

5.2.1.1 Standard operation

For each macro time step, computational complexity is proportional to the number of modules receiving input. The management unit stores them in an ordered graph. At each macro time step, first the input from outside the system is feed to abstraction modules directly connected to it. Then, their results are fed on through the graph.

Modules indicate to the management unit whether they produced new output upon the input. E.g., a module evaluating $a > 0$ will not produce new output when a changes from 4 to 5. This is important because in practical abstraction graph, the data volume output by modules is a fraction of the input volume for most modules.

$$C = O(n)$$

with	C	computational complexity
	n	number of modules receiving new input

Storage requirements are defined by the management unit keeping track of each module, and the sum of the requirements of all modules.

$$S = O(m) + \sum_{i=1}^{m} S_i$$

with	S	storage requirement
	m	number of modules
	S_i	storage requirement of module i

5.2.1.2 Internal playback mode

Asbru permits to specify the parts of the time annotation by arbitrary expressions. In practice, constants are generally used, but the possibility to use variables requires the following provisions. If a non-constant reference is found in the Asbru elements specifying the parts of the time annotation, then an internal buffer is introduced which stored the input to the ParameterPropositionModule. As soon as the time annotation is evaluated according to the Asbru semantics, the playback of the stored values is triggered. This means that these values are fed to the ParameterPropositionModule and only the last output is forwarded to the modules further down the abstraction graph.

This leads to extra computational effort in a small fraction of time steps. The effort is proportional to the number of stored data points.

$$\begin{array}{rcl} C & = & O(p\,h) \\ \text{with} & C & \text{computational complexity} \\ & p & \text{number of modules requiring playback} \\ & h & \text{length of history, i.e., preserved input} \end{array}$$

Although the likelihood that playback is needed for any abstraction module at a given time step is small, there is no guarantee that not all of the concerned modules will require it simultaneously.

The length of history depends on the frequency at which data points needing storage arrive, and the delay between program start and evaluation of the time annotation. Both depend heavily on the model (i.e., the Asbru plan library).

The storage required is also proportional to the number of modules requiring playback multiplied by the (average) length of the recorded history.

$$\begin{array}{rcl} S & = & O(p\,h) \\ \text{with} & S & \text{storage requirement} \\ & p & \text{number of modules requiring playback} \\ & h & \text{length of history, i.e., preserved input} \end{array}$$

5.2.1.3 Parent-child communication

When a child plan changes its state, the module implementing it sends a PlanState-DataPoint to the module implementing the parent. The valid time of this data point is the next micro time step, since this communication is directed “against the flow”. The communication from the parent to the child happens “with the flow”, i.e., in the normal direction of the graph formed by all the modules.

It is possible that the state change of the child plan causes a state change of the parent. In this case, a chain of PlanStateDataPoint transmissions results. Each link of this chain is executed in another micro time step. Micro time steps do not depend on external timing. Since there are very few data points to process (generally only one), the computational effort per micro step is small. Nonetheless, it is considered here, in b being the number of plans which change their state at this micro time step and therefore generate another PlanStateDataPoint.

The maximum number of micro steps depends on the depth of the plan hierarchy, i.e., on the number of generations from the top-level plan to its furthest child.

$$C = O(d\,b)$$

with	C	computational complexity
	d	depth of the plan hierarchy
	b	number of back-linking modules, i.e., plans changing their state simultaneously

There is not extra storage required by this issue.

5.2.1.4 Summary

In total, the computational complexity and the required storage space are the respective sums of the three parts described above.

$$C = O(n) + O(p\,h) + O(d\,b)$$

$$S = O(m) + \sum_{i=1}^{m} S_i + O(p\,h)$$

with	C	computational complexity
	n	number of modules receiving new input
	p	number of modules requiring playback
	h	length of history, i.e., preserved input
	d	depth of the plan hierarchy
	b	number of back-linking modules, i.e., plans changing their state simultaneously
	S	storage requirement
	m	number of modules
	S_i	storage requirement of module i
	p	number of modules requiring playback
	h	length of history, i.e., preserved input

5.2.2 Utility functions

Modules implementing utility functions can be grouped by their number of inputs, which can be zero, one, two, or any number. Both computational effort and storage requirement are proportional to the number of inputs.

There is only one exception in this group: Abstracting qualitative values based on numeric input. Here, there is only one input, the complexity is proportional to the number of values which the qualitative output can take.

$$\begin{aligned} C = S &= O(1) \quad \text{if } i \leq 1 \\ C = S &= O(i) \quad \text{otherwise} \end{aligned}$$

with	C	computational complexity
	S	storage requirement
	i	number of inputs of this module, number of qualitative levels in case of qualitative abstractions

5.2.3 Sliding time windows

The modules presented under this heading are grouped into those collection data and those evaluating the content of the first group.

5.2.3.1 Collection of data

There are three types of time windows. The number of time points they contain varies accordingly:

- Interval-based time window. The maximum number of data points within a time window is given by the (maximum) input frequency times the length of the window.
- Time window based on number of measurements. Here, the maximum number of data points is directly stated.
- Episode-based time window. The number of data points depends on the length of the episode, i.e., the interval matching a given time annotation. There is no formal limit on the duration of an episode. In practical applications, the time annotation will impose some limit, if the – often implicit – clinical knowledge is encoded properly.

Time windows can be sorted by transaction time, i.e., the time of arrival of the data point, or by value of the data point. In the first case, the effort is only linear. In the second, it depends on the sorting algorithm. Since steadily increasing measurements, which lead to a worst case scenario for Quicksort, are often seen in practice, I assume quadratic effort.

Output occurs at a reduced frequency. Only at these times, sorting is performed. The effort of adding the input and discarding old data is constant.

$$\begin{array}{rcll} C & = & O(1) + O(1) + \frac{O(1)}{w} = O(1) & \text{if sorted by transaction time} \\ C & = & O(1) + O(1) + \frac{O(p^2)}{w} = O(p^2) & \text{if sorted by value} \\ S & = & O(p) & \end{array}$$

with	C	computational complexity
	p	number of points in the time window
	w	step width of data output, in time steps
	S	storage requirement

The modules analyzing time windows can be grouped according to their complexity as shown in the following subsections.

5.2.3.2 Analyzing the time window content

Based on sorted time windows, the following parameterless statistical measures can be performed with constant effort: median, minimum, maximum, centiles and change (between the first and the last measurement. Also accessing temporal properties start, end, and duration is performed with constant effort.

The effort for the calculation of average (normal and weighted by time for which the measurement is valid), as well as standard deviation within a time window is proportional to the number of data points in the time window.

The effort to calculate the linear regression line for a given content of a time window is proportional to the number of data points in the window. This comprises calculation all of its features including standard error. Accessing these properties (slope, standard deviation and error, end point, centre, time to alarm) is performed with constant effort.

$$\begin{array}{rcll} C & = & O(1) & \text{for median, minimum, maximum, centiles and change;} \\ & & & \text{start, end, and duration of time window;} \\ & & & \text{and accessing linear regression features;} \\ C & = & O(p) & \text{for average, standard deviation, linear regression;} \\ S & = & O(1) & \text{in all cases} \end{array}$$

with	C	computational complexity
	S	storage requirement
	p	number of points in the time window

5.2.4 Coping with noisy and missing data

The LimitModule compares the input against two limits, at constant computational effort. All other modules which can be combined to implement custom solutions to detect errors in the input have already been discussed above.

The Spread algorithm consists of the following parts.

Sliding time window	$C = O(1)$	$S = O(p)$
Linear regression	$C = O(p)$	$S = O(1)$
Qualitative value abstraction	$C = O(v)$	$S = O(v)$

with	C	computational complexity
	S	storage requirement
	p	number of points in the time window
	v	number of possible qualitative values

The only new part is the third. Mapping the Spread margin to a qualitative value is proportional to the total number of possible values which the output can take, both in computational complexity (going through the limits comparing them against the quantitative value) and in storage requirement (to store the limits).

The computational effort differs significantly between time steps at which output is produced and such where none is produced.

The total effort and requirements for the Spread algorithm are:

$$\begin{array}{rcll} C &=& O(p)+O(v) & \text{for every } w^{th} \text{ time step} \\ C &=& O(1) & \text{for other time steps} \\ S &=& O(p)+O(v) & \end{array}$$

with	C	computational complexity
	S	storage requirement
	p	number of points in the time window
	v	number of possible qualitative values
	w	step width at which output is produced

5.2.5 Online-algorithms for the detection of temporal patterns

5.2.5.1 Parameter propositions

Monitoring parameter propositions is implemented in multi-dimensional state machines which perform zero to two state transitions per time step, resulting in a computational effort and storage requirement in the order of $O(1)$.

$$C = S = O(1)$$

with	C	computational complexity
	S	storage requirement

5.2.5.2 Temporal relations

Monitoring temporal relations is summarized in Table 5.1. For different relations, different combinations of lists are maintained.

Both computational effort and storage requirement depend on the value u which is the length of current output list. This is the number of concurrently valid matching pairs of intervals, i.e., a rather small number in practice, yet for relations *before*, its theoretical upper bound is ab or $O(a^2)$ assuming a and b being of the same order of magnitude. The reason for this high upper bound is the rare case where all occurrences of A lie before all occurrences of B. For the other relations, the upper bound of u is $O(a)$.

Because of this, the storage requirement can be traced back to $O(a)$, too.

$$\begin{array}{rcll} S = C &=& O(a^2) & \text{for } \textit{A before B} \\ S = C &=& O(a) & \text{for other relations} \end{array}$$

with	C	computational complexity
	S	storage requirement
	a	number of intervals A in the past

5.2.5.3 Boolean combinations

Monitoring Boolean combinations of intervals is detailed in Table 5.2. The algorithm monitoring the *or*-disjunction runs through the output list and possibly adds entries to two other lists. For each entry in both, it then runs through the output list again. Therefore, the worst case computational effort is $O(u^2)$.

Relation name	Computationally worst case	Computational effort	Storage requirement
A before B	A revoked	$C = O(a) + O(u)$	$S = O(a) + O(u)$
A overlaps B	A or B revoked	$C = O(a) + O(u)$	$S = O(a) + O(u)$
A starts B	A or B revoked	$C = O(u)$	$S = O(u)$
A equals B	A or B revoked	$C = O(u)$	$S = O(u)$
A meets B	A or B revoked	$C = O(u)$	$S = O(u)$
A during B	A or B revoked or $\mathrm{NF_B}$	$C = O(u)$	$S = O(u)$
A finishes B	A or B revoked	$C = O(u)$	$S = O(u)$

Table 5.1: Computation effort and storage requirement for monitoring temporal relations.
a is the number of occurrences of interval A in the past. This is assumed to be in the same order of magnitude as the number of past interval B occurrences.
u is the length of current output list.
Event "A revoked" means an occurrence of A is no more valid. Likewise for B.
Event $\mathrm{NF_B}$ designates the end of an occurrence of B.

$$\begin{array}{rcll} C &=& O(u^2) & \text{for } \textit{or} \text{ and } \textit{xor} \\ C &=& O(u) & \text{for } \textit{and} \text{ and } \textit{not} \\ S &=& O(i) + O(u) & \text{for } \textit{and} \\ S &=& O(u) & \text{for other relations} \end{array}$$

with	C	computational complexity
	S	storage requirement
	i	number of inputs
	u	length of output list, i.e., number of concurrently valid matching intervals

5.2.5.4 Extracting features of episodes

A set of utility modules is provided which all operate at constant (and minimal) effort without requiring storage for themselves.

$$C = S = O(1)$$

with	C	computational complexity
	S	storage requirement

5.2.6 Integration of plan execution

Each module representing an Asbru plan – or a single plan step in it – is a finite state machine representing the Asbru plan states. The number of state transitions at any time step is limited to four by the semantics of Asbru (from initial state via *considered*, *possible*, *active* to either *completed*, *aborted* or *suspended*). Cycles are only

Relation name	Computationally worst case	Computational effort	Storage requirement
and	i revoked	$C = O(u)$	$S = O(i) + O(u)$
	$\mathrm{PF_i}$	$C = O(i)$	
or	$\mathrm{PF_i}$ or $\mathrm{ID_i}$ revoked	$C = O(u^2)$	$S = O(3u) = O(u)$
xor	as above	$C = O(u^2)$	$S = O(u)$
not	in revoked	$C = O(u)$	$S = O(u)$

Table 5.2: Computation effort and storage requirement for Boolean combinations of interval series.
i is the number of inputs to the combination.
u is the length of current output list.
Events "i revoked" and "$\mathrm{ID_i}$ revoked" means that a previously found interval for input channel i is no more valid.
Event $\mathrm{PF_i}$ designates the start of a *true* value at input channel i.

possible between *active* and *suspended* and they are prohibited to occur in the same time step, both by the semantics and the implementation. Therefore, the upper limit of computational effort is constant.

The plan modules only keep a set of status variables, some of which reflect the state of child plans. Therefore, the storage requirement depends on the (small) number of child plans (also called subplans).

$$\begin{array}{rcl} C = S & = & O(1) \\ \text{with} & C & \text{computational complexity} \\ & S & \text{storage requirement} \end{array}$$

$$\begin{array}{rcl} C & = & O(1) \\ S & = & O(c) \\ \text{with} & C & \text{computational complexity} \\ & S & \text{storage requirement} \\ & c & \text{number of child plans} \end{array}$$

5.2.7 Bridge from Asbru to abstraction modules

Transforming the Asbru plan library into a graph of abstraction modules takes place before monitoring the input is started. Therefore, it does not contribute to the computational complexity of the data abstraction and plan execution process.

Likewise, storage required for parsing the XML file containing the Asbru plan library is released before the monitoring is started.

Therefore, no extra burden is added by this part of the solution.

The transformation of the plan library into a set of abstraction modules is proportional to the number of modules.

C	$= O(1)$	sliding time window aggregation if sorted by transaction time; feature extraction for time window, episodes, and linear regression; monitoring parameter propositions in the normal case; plan execution;
C	$= O(i)$	utility functions;
C	$= O(p)$	average, standard deviation, linear regression; Spread algorithm;
C	$= O(a)$	temporal relations other than *A before B*; monitoring parameter propositions at the moment when playback is required;
C	$= O(u)$	Boolean combinations *and* and *not*
C	$= O(p^2)$	sliding time window aggregation if sorted by value;
C	$= O(a^2)$	temporal relation *A before B*;
C	$= O(u^2)$	Boolean combinations *or* and *xor*;
with	C	computational complexity
	i	number of inputs
	p	number of points in the time window
	a	length of history
	u	number of concurrently valid matching intervals

Table 5.3: Summary of computational effort of modules. The number of possible values on a qualitative scale is assumed to be much smaller than the number of data points in a time window.

5.2.8 Discussion

The computational complexity varies significantly between classes of modules. In particular, many of the frequently used building blocks have constant or linear effort functions, while some have quadratic functions. None of them are NP-hard.

Table 5.3 groups the modules by computational effort.

In practice, the nastiest case are those cases, where the effort depends on the size of the history, since this can rapidly grow in high-frequency domains. There are two simple measures to reduce the problem which are natural in care practice:

- The frequency of the input to the computational expensive modules is drastically reduced by aggregation and abstraction modules which turn the input stream of high-frequency measurement into a series of rather few changes in clinically meaningful concepts. While the reduction cannot be quantified in a general way, a strong guarantee is given by the fact that unstable values are not considered a meaningful basis of clinical decision making. Sometimes, minimum requirements are even explicitly stated, such as “high for 3 minutes”, which means that the frequency of such a datum cannot exceed 20 per hour.

- There is a temporal limit for the clinical meaningfulness of measurements. This means, that not all episodes in history are considered, but only those in a limited time window. Whether this comprises the most recent hour or day or year depends on the domain, but since human perception cannot deal with large quan-

S	$=$	$O(1)$	analysing time window content, including linear regression; feature extraction for time window, episodes, and linear regression; monitoring parameter propositions;
S	$=$	$O(i)$	utility functions;
S	$=$	$O(c)$	plan execution;
S	$=$	$O(p)$	sliding time window aggregation; Spread algorithm;
S	$=$	$O(a)$	temporal relations other than *A before B*;
S	$=$	$O(u)$	Boolean combinations other than *and*;
S	$=$	$O(i) + O(u)$	Boolean combination *and*;
with		S	storage requirement
		i	number of inputs
		c	number of child plans
		p	number of points in the time window
		a	length of history
		u	number of concurrently valid matching intervals

Table 5.4: Summary of storage requirements in modules. The number of possible values on a qualitative scale is assumed to be much smaller than the number of data points in a time window.

tities of information, domain experts tend to specify their abstraction rules in a way which reduces the amount of data in the time window in a very favourable way.

Table 5.4 gives an overview of the storage requirements. For those values which threaten to grow over time – a and u – the precautions described above can be taken.

5.3 Meeting the Objectives

In Section 3, I described the objectives for a system which qualifies as an answer to the research question. In this subsection, I detail how the solutions presented in Section 4 meet these objectives.

> **Objective 1.** To combine existing approaches such as comparing different input channels, averaging and comparing against the average, into the framework for data abstraction, error detection and repair.

Sections 4.2 through 4.4 provide a rich toolset for the detection of errors and the repair of the gaps.

In particular, Section 4.2.1 describes the following operations directly performed on numeric input: sum, product, minimum, or maximum for any set of parallel inputs; difference, quotient, root, exponent, and logarithm for pairs of input; and absolute value and sign for a single input. Comparison operators and Boolean logics on their results are implemented as shown in Section 4.2.3.

Section 4.3, also mentioned at Objective 3, provides means to collect measurements in a sliding time window and to calculate change over time and various means of deviations within them.

Section 4.4.1 describes the combination of the above mentioned modules for error detection. Section 4.4.2 describes means for data repair. The examples given in these two subsections illustrate how complex requirements can be broken down and translated into a combination of modules from the available toolset.

The framework integrating these modules is described Section 4.1.

> **Objective 2.** To find a new solution for abstracting steady qualitative values from quantitative data containing varying parts of noise, such that the threshold used to suppress undesired changes in the output depends on the current amount of noise.

The Spread algorithm described in Section 4.4.4 continuously adapts the threshold for mapping noisy, quantitative values to steady, qualitative abstractions. It comes in different versions (e.g., based on standard deviation or quantiles) to adapt to different types of signal sources and numerous parameters provide rich options for fine-tuning. It performed very satisfactory in a series of practical clinical studies described in Section 5.1.1.

The set of time window property queries described in Section 4.3 permit explicit reasoning about the data quality during the monitoring process. This can be used to switch between different modes of abstractions, e.g., to provide an additional fallback strategy for phases of extreme noise, or to pick between different sources of input, depending on their current quality.

> **Objective 3.** To integrate the aggregation of input in sliding time windows of freely defined size and step width into the common framework, together with evaluation functions such as average, mean, quantiles, and linear regression.

Section 4.3 describes how values can be aggregated for different types of time windows: time-oriented, measurement-count oriented, or episode-oriented, where episode-oriented refers to an state of an external condition (fulfilled or not fulfilled) delineates the time window.

Section 4.3.2 describes abstractions based on these time windows: median, minimum, maximum, average, or standard deviation, change between end points of the time window, time-oriented average, centiles.

Section 4.1 describes the integration of all these modules into a uniform framework.

> **Objective 4.** To integrate utility functions implementing arithmetic and logical operators as well as comparison operators with the other modules.

While trivial for themselves, the provided utility functions such as computations and extraction of features from complex data items significantly increase the usability of the more complex abstraction algorithms. Their free combination permits the modeller to add arbitrarily complex complements to the specialised abstraction modules and the plans.

> **Objective 5.** To integrate all the algorithms presented in this thesis in a framework permitting the greatest possible freedom in combining them. Data flow in this framework must be organised in such a way as to minimise computational effort.

Section 4.1 describes a framework containing the modules described in this thesis. Integrating data abstraction, monitoring and plan execution in a single, seamless framework greatly reduces system complexity while providing new options, e.g., to feed plan output back into abstraction modules.

The fact that any computation is only performed as mandated by new input or previously set alarms permits the execution of complex plan libraries at high frequency on standard computers.

> **Objective 6.** To find implementations for the Asbru elements *parameter-proposition*, *plan-state-constraint*, *temporal-constraint*, *constraint-combination*, *count-constraint*, and *simple-condition*. They must be integrated into the framework of temporal data abstraction and guideline execution, and handle high volumes of data efficiently.

Integrated in the framework described in Section 4.1, the algorithms presented in Section 4.5 bridge temporal data abstraction to plan execution with minimal computation effort. This is achieved by implementing the complex matching rules in the form of state machines. Only when new data arrives, simple rules are evaluated in the modules concerned and where suitable, the internal state of a monitoring module is changed.

Table 5.5 shows which section describes which solution in detail, and the computational effort for each of the Asbru elements listed in the objective.

> **Objective 7.** To translate the semantics of Asbru plans to the process-logic the abstraction framework.

Asbru element	Section	Computational effort	
parameter-proposition	4.5.2	$O(1)$	
plan-state-constraint	4.5.3	$O(1)$	
temporal-constraint	4.5.4	$O(a)$	for temporal relations other than *A before B*
		$O(a^2)$	for temporal relation *A before B*
constraint-combination	4.5.5	$O(u)$	for Boolean combinations *and* and *not*
		$O(u^2)$	for Boolean combinations *or* and *xor*
count-constraint	4.5.6	$O(1)$	
simple-condition	4.2.3.6	$O(1)$	
with	C	computational complexity	
	p	number of points in the time window	
	a	length of history	
	u	number of concurrently valid matching intervals	

Table 5.5: Implementation of Asbru elements from Objective 6.

While the original design of Asbru foresaw querying the state of conditions at a fixed sampling rate, the presented framework enacts all plan state changes immediately as soon as the input changes. Besides removing any delays, it is also more efficient than repeated polling by all plans.

Section 4.7 describes the Asbru semantics and how they are implemented by modules. These modules communicate among each other using PlanDataPoint objects which are compatible with those used to communicate values between abstraction modules. This allows the Management Unit of the abstraction framework to handle the information flow between parent and child plan in the same fashion as the data flow within that part of the module network which implements the temporal data abstraction.

Objective 8. To map all Asbru elements relevant for plan execution to one or more modules implementing the semantics of this element.

This thesis describes the implementation of the complete Asbru functionality with the exception of the setup-precondition and complex data structures. Their omission has been justified by their redundancy in practical modelling work.

The projects described in Section 5.1 showed that the implemented subset is sufficient model the protocols and guidelines used in these projects (US guideline for Jaundice in neonates, Dutch guideline for breast cancer, Italian protocol for medical treatment of breast cancer, management of birth, emergency reception of stroke patients, management of MRSA infection, management of drug administration for the elderly at an emergency department).

Details on the mapping between Asbru and the modules described before are found in Section 4.7.

5.4 Limitations

5.4.1 Coping with noisy data

In practical applications, we found that the major limitation is the acquisition of precise information about the acceptable delays and fine-grained quality measurements for the quality of the abstraction. Practitioners generally did not reason about such a technical perception of the patient. The only gold standard is the overall outcome, but this can be achieved in many different ways.

A minor technical limitation of the described system is the lack of advanced abstraction methods from Statistics and Machine Learning. It is easy to write a wrapper which translates output from this framework to input for systems like R or Weka. Results from there can be fed back into this framework as parameters.

5.4.2 Online-algorithms for monitoring temporal patterns

There are some limitations in the expressive power and precision of the Asbru syntax. First, time annotations have only one reference point. It is therefore not possible to define the end of an interval based on another external reference as the start.

Second, temporal pattern can freely combine complex constraints on episodes in the input data, but it cannot define two constraints to be applied on exactly the same instances in the stream of episodes because there is no way to refer to individual instances, only to their class.

5.4.3 Multiple sliding time windows for statistical analysis

It lies in the nature of sliding time windows that the statistical evaluation is performed more often then for fixed time windows. This may lead to unacceptable delays in the analysis of large data sets at fine granularity. However, in practical application, it proved easy to choose a step width large enough to avoid such delays.

While performance issues generally do not limit the amount of parallel and diverse abstractions based on many different time windows, the acquisition of the medical semantics of the output proved to be an important limiting factor.

5.4.4 Integration of plan execution

The functionality of the setup-precondition was not implemented. The Asbru design prescribes that the execution unit autonomously searches suitable plans to fulfil this condition, based on the plan effects described in the plan library. However, a use case for this functionality never appeared.

One reason for this is that decisions between alternatives can easily be made explicit for systems which are not too complex. The other reason is that clinical guidelines and protocols seem to be written with such explicit information in mind. Finally, it is always much more difficult to acquire the hidden knowledge behind the prescribed processes, than to acquire the direct formulation for the conditions under which a certain action is performed. The latter goes to the filter-precondition, which is the most often used condition in all projects, while the setup-precondition together with plan effects remained generally unused.

5.4.5 Utility functions

List manipulation was not implemented in the interpreter and is not discussed here. It was introduced during the creation of Asbru 7, but practical demand for it did not appear in the decade thereafter.

5.4.6 Uniform framework

The information flow between the presented system and the user interface needs some further development. The OncoCure project showed demand to synchronize requests, presenting alternatives together in one request, and generally presenting more information than currently supplied by the interface.

Besides this, the current implementation is optimized for high-frequency domains and batch processing of protocols based on recorded data. Consequently, the management unit starts up at program start – before the execution of the guideline – then performs all steps of the protocol, terminating after completion of the protocol.

With low-frequency domains in mind, the same ideas presented in this thesis could and should be implemented as a web-service with a database back-end which performs one step at a time, reading and writing all information to and from a database. This is more suitable for domains where few processing steps occur during a patient encounter and nothing happens for many weeks between these encounters.

5.4.7 Bridge to Asbru

Complex data structures as described in Asbru 7.4 were not implemented. They could be added within the framework presented in this thesis. However, they were controversial when introduced and may not be part of the next major language version.

Consequently, the *for-each* plan and the *iterative* plan were omitted in this thesis, because they are focused on list processing.

Chapter 6

Future Directions of Work

6.1 Further development of the system

On a technical level, there are two directions of further developing the system presented in this thesis: Adding abstraction modules, and better integration of user interaction.

6.1.1 Expansion of the set of abstraction algorithms

Numerous as they are, the currently implemented algorithms for temporal data abstraction are a small fraction of the known corpus of useful algorithms in this field. To avoid the creation of yet another mighty but hard to learn toolset, this extension process will need to be application-guided.

6.1.2 Better control over user interaction

The design presented in this thesis focuses on the efficient automatic processing of large volumes of data, touching user interaction only slightly. Mostly in low-frequency domains, but also in some high-frequency applications, better control over user interaction and presentation of options is desired. Implementing this without breaking the overall structure of the framework (and thus loosing its advantages) poses an interesting challenge for the future. It also prompts changes in the representation of the plan library, i.e., the syntax of Asbru.

6.2 Neighbouring fields of research

As shown in Chapter 1, the ideas presented here can only be a small part of the solution. Our work showed clearly that there are pressing issues in fields related to this work, which prevent the wider application of the solutions presented here. Most notably, modelling and data integration are bottlenecks in the development of computer-aided application of guidelines.

6.2.1 Modelling

All three projects described in Section 5.1.2 showed that modelling a guideline requires international collaboration.

Since physician time is scarce and computer scientist time lesser so, and also since physicians are not trained to create abstract models, it is frequent practice that the computer scientist creates the best approximation of the solution based on the available information, and then discusses it with the physician. This must be repeated many times, due to the complexity of the domain and the short time slots available in the daily life of physicians.

A long-standing effort to reduce this problem is the creation of representations which are meant to be easily understood by persons without IT training. Our experiences in evaluating MHB [129] showed that a prerequisite for success in this pursuit is an editing environment with embedded just-in-time learning facilities.

In my next project, I will try to answer these challenges.

6.2.2 Data integration

The guideline modelling community focuses on this issue for the recent decade. Unfortunately, progress is hard to obtain given the large number of powerful, and not too flexible, players. Standards for patient data slowly evolve, and as they do, implementations must ensure that they can work with them.

In the case of the Asbru interpreter, mapping the parameter names to references in a standard terminology or ontology can already be done, but a wrapper to an existing system has not yet been implemented.

Chapter 7

Conclusions

Clinical guidelines and protocols are an important means to improve the quality of care. To make their application more efficient, they are translated to computer-interpretable models using languages such as Asbru, which represents treatment as hierarchy of plans. Temporal data abstraction is required to bridge the low-level data from monitoring devices and laboratory results to the high-level concepts used in clinical guidelines and protocols.

The main research question answered in this thesis is:

> How can temporal data abstraction be combined with the execution of clinical guidelines and protocols in a fashion suitable for high-frequency domains?

This question brings along the following subquestions. The combination of my answers to them constitutes my answer to the main research question on an abstract level. On a detailed level, I described a solution building on these answers in Section 4.

> **Subquestion 1:** How can short response times be established for arbitrarily high volumes of data?

The nucleus of my answer is to create a push network of modules each of which implements a fraction of the overall functionality. Modules are only activated when new input for them is available. This means that the computational effort is either the specific effort of the module concerned, for a minority of modules, or zero, for the majority.

The claim that the majority of modules does not receive new input in any given time step cannot be specified mathematically. It is based on the common modelling practices of knowledge engineers who are aware of performance aspects. However, in the medical domain, it is common to abstract high-frequency data into lower-frequency abstractions, because they are more tangible to human reasoning and expression. Therefore, it is only a natural depiction of the domain knowledge to transform high-frequency signals into lower-frequency streams of information in the first steps into the network.

An important condition for the above is that this abstraction is not just simple averaging, but modellers can pick from a rich list of alternatives. This is established in the system described in Section 4 by providing a wide range of abstraction modules which can be freely combined by the knowledge engineer.

> **Subquestion 2:** How can steady values be abstracted from qualitative input with varying amounts of noise and gaps of varying length?

The answer to this question needs to differentiate between coping with noise and gaps in the input data on the one hand; and abstracting quantitative values and abstracting qualitative values on the other hand.

Noise in the measured signal can be separated in cases which can be isolated, and those which cannot. In the first case, rules to detect them are part of the domain knowledge. These can be implemented using rather simple abstraction modules. The key here is the free combination of a range of modules, to permit the knowledge engineer to tailor the rules to the specific characteristics of the signal. In the second case, it is treated differently in the following depending on the quantitative or qualitative nature of the abstractions derived from such data.

For *quantitative* values, statistics provides a range of accepted solutions. In medicine, considering the median instead the mean is a popular and very simple solution. More sophisticated but still computationally well affordable is the following: Compute a linear regression model of the measurements in a certain time window. Then remove those for which the square of the residuals (i.e., the vertical distance between the data point and the regression line) exceed the squared standard deviation multiplied by a chosen factor.

The statistical approaches are complemented by knowledge-based approaches, where domain knowledge is encoded into consistency checks between multiple input channels. This permits to draw a more precise line between distorted and undistorted inputs in those cases where such domain knowledge is available.

To close gaps, interpolation and extrapolation based on a suitable model of the underlying function can be used. In practice, linear regression models provide a fair approximation at limited computational cost. Again, domain knowledge can be used to anticipate changes which are not expressed in the data, but which are given by the nature of the measured entity. E.g., the slope of a value can be known to become level in the normal region for some signals, or maxima like 100% saturation cannot be exceeded.

In the context of combining temporal data abstraction with guideline execution, extrapolation is rarely used. Instead, gaps in the input data raise alarms which call for higher-level action, like starting an emergency plan for disconnected sensors, instead of speculating about the possible value of an unknown input.

For *qualititive* values, an important target is to obtain steady values. I.e., once the border between two qualitative values is crossed, then the output should remain in the other region for some time. At the same time, delays in showing important changes must be avoided. The general answer to this challenge is the introduction of a hysteresis, or threshold for changing to the next qualitative value. Unfortunately, a fixed threshold is only suitable for fixed amounts of undesired oscillation in the input.

Consequenly, I introduced variable thresholds, which are based on the oscillation of the signal in the observation period. Statistically justified calculations for these thresholds are the standard deviation, standard error, and the distance of centiles to the mean.

Specifically, in the Spread algorithm, a linear regression model is calculated for a moving time window. On the end point of the line, i.e., at the last measurement, or current time point, the standard error is plotted up and down. Only if the whole length of the resulting bar passes the horizontal line which denotes the border between two qualitative regions, the qualitative output value is changed. This generates a memory or delay effect which is the bigger, the bigger the oscillations are.

An alternative design of the Spread is to draw the bar from the 10% centile to the 90% centile (or any other pair of centiles). This proves useful for asymmetric signals, where the deviations to the lower side differ significantly from those to the upper side.

Besides the amount of oscillation, the delay time is controlled by the centile chosen, or a factor multiplied with the standard deviation or error.

In cases where the delay effect is unacceptable and there is a *normal* value defined in the domain knowledge, the following variant proved stable enough in many cases: The qualitative value is chosen from that region where the margin nearer to the normal value lies.

All of the above approaches are based on sliding time windows which aggregate the measurements of the most recent minutes, with the length configured by the knowledge modeller. Linear regression implicitly fills gaps in the input data, but domain knowledge will define minimum amounts of data required for reliable estimates of the measured value. Thanks to the arbitrarily complex network of abstraction modules, rules such as "the output is considered missing if more than 50% of the input values in the last 5 minutes or more than 20% in the last minute are invalid".

Since smoothness in qualitative values is a desire property, filling gaps by means of linear interpolation is the appropriate method in most cases. Non-linear interpolation is not precluded by the system design. Summing up, it can be said that the problem of filling gaps is reduced to the problem of defining measures for the acceptability of the amount of filling performed.

> **Subquestion 3:** How can the execution of Asbru plans be combined with temporal data abstraction in an efficient way?

My answer to this question is to translate the semantics of Asbru to a set of state charts which describe all aspects of plan execution. This way, they can be combined with the push network of data abstraction modules in a way as efficiently as the data abstraction is implemented.

For plan execution, the gains from the push architecture are even bigger than for temporal data abstraction. The original Asbru description foresaw repeated queries of the patient state, which inevitably must lead to futile processing steps and delays in reaction. In the implementation described in this thesis, plan execution only consumes computing time if there is a chance to change any plan state. At the same time, changes in input data take effect without any delay.

Besides transforming the declarative semantics of plans and environment monitoring into state machines compatible with the operation of the data abstraction network, I had to devise a set of abstract data points to transport the sometimes complex chunks of information between these modules implementing monitoring and plan execution.

Containing a valid time like input data, they are compatible with the time-stamped data points passed on between abstraction modules. Various property access methods are used to extract simple numbers like the duration of found intervals out of complex entities like the group of intervals matching a certain temporal pattern.

The design of an efficient push network for temporal data abstraction, the provision of a wide range of abstraction modules to filter noise where possible and to handle that which cannot be filtered out, and the translation of Asbru syntax to state machines compatible with the abstraction network altogether forms my answer to the question "How can temporal data abstraction be combined with the execution of clinical guidelines and protocols in a fashion suitable for high-frequency domains?"

As discussed in Section 5.2, the computational effort is proportional to the number of modules with a few exceptions. These exceptions are Boolean combinations of intervals and temporal relations between intervals which by their nature demand effort which is proportional to the length of the history under consideration, or its square. The rich collection of abstraction modules permits the knowledge engineer to reduce the length of the history on which the computational effort depends, to the time window which a human expert would examine, thereby cutting out many combinations of intervals which are only of theoretical value.

The evaluation of the work presented here was performed on a theoretical and a practical level. On the theretical level, I presented the computational effort and the storage requirements of the algorithms I introduced. I also describe their limitations, and who they meet the objectives set forth in this thesis.

On the practical level, my solutions are fundamental in two lines of application. On the one hand, the abstraction of steady qualitative values from noisy data permitted the control of oxygen supply in a neonatal intensive care unit at the level of a human expert dedicated to the job, and superior to clinical routing.

On the other hand, the integrated framework for temporal data abstraction, monitoring, and plan execution forms the basis of the Asbru interpreter which was (and is) used successfully in three international research projects.

While this thesis focuses on medicine, more specifically intensive care, the described methods can be applied to other fields with one or more of the following properties: (1) temporal data abstraction is combined with plan execution; (2) data is heterogeneous, voluminous, and time-stamped; (3) noise is an issue; and/or (4) the modelled knowledge is complex.

Acknowledgements

I honestly wish to thank the following persons:

Werner Horn for making my master thesis part of a success story, and Silvia Miksch for recruiting me for the Asgaard project. Without both together, I would never have started the work described in this thesis.

Katharina Kaiser for being such a great mediator.

Everyone involved in the Pulsoximetry project for making it a success.

Claudio Eccher for taking the Asbru interpreter to clinical practice.

Michael Paesold for his valuable feedback.

All the nice people I met in numerous countries and projects for the good times we had.

Isabella Seyfang for my strength.

Bibliography

[1] Health level seven arden syntax for medical logic systems, version 2.1 ANSI/HL7 Arden V2.1-2002. Technical report, 2002.

[2] ASTM standard E2210-06. Standard specification for Guideline Elements Model version 2 (GEM II). Document model for clinical practice guidelines. ASTM International, West Conshohocken, PA, 2006.

[3] Electronic statistics textbook. Technical report, StatSoft, Inc., 2010. Available at http://www.statsoft.com/textbook/, last accessed October 20$^{\text{th}}$, 2010. Also available in print: [71].

[4] Free Merriam-Webster dictionary: Noise, 2010. Available at http://www.merriam-webster.com/dictionary/noise, last accessed September 22$^{\text{nd}}$, 2010.

[5] Impact of oncodoc2 on guideline compliance in the management of breast cancer. trial description at clinicaltrials.gov, 2010. Available at http://clinicaltrials.gov/ct2/show/NCT00728442, last accessed October 14$^{\text{th}}$, 2010.

[6] InferMed product information on Arezzo, 2010. Available at http://www.infermed.com/index.php/arezzo/arezzo_technology last accessed October 11$^{\text{th}}$, 2010.

[7] Object constraint language, version 2.2. Technical report, Object Management Group, 2010. Available at http://www.omg.org/spec/OCL/2.2, last accessed October 11$^{\text{th}}$, 2010.

[8] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, pages 832–843, 1983.

[9] L. Anselma and S. Montani. Planning: Supporting and optimizing clinical guidelines execution. In ten Teije et al. [158], pages 101–120.

[10] M. Balaban, D. Boaz, and Y. Shahar. Analysis of temporal abstraction in medical databases. In *Knowledge Representation meets Databases (KRDB)*, 2003.

[11] M. Balser, C. Duelli, and W. Reif. Formal semantics of Asbru - an overview. In *Proceedings of the 6th biennial world conference on integrated design and process technology (IDPT-02)*, page 1, 2002.

[12] B. Beauregard. Oracle database 11g workspace manager overview. Technical report, Oracle Corporation, World Headquaters, 500 Oracle Parkway, Redwood Shores, CA 94065, USA, September 2009. Available at http://www.oracle.com/technetwork/database/twp-appdev-workspace-manager-11g-128289.pdf, last accessed October 29th, 2010.

[13] M. Beccuti, A. Bottrighi, G. Franceschinis, S. Montani, and P. Terenziani. Modeling clinical guidelines through petri nets. In *Artificial Intelligence in Medicine*, volume 5651, pages 61–70, 2009.

[14] S. Y. Belal, A. F. G. Taktak, A. Nevill, and A. Spencer. An intelligent ventilation and oxygenation management system in neonatal intensive care using fuzzy trend template fitting. *Physiological Measurement*, 26:555–570, 2005.

[15] R. Bellazzi, C. Larizza, and G. Lanzola. An HTTP-based server for temporal abstractions. In *IDAMAP 1999 working notes*, pages 52–62, 1999.

[16] S. Bernardi, S. Donatelli, and A. Horvath. Implementing compositionality for stochastic petri nets. *International Journal on Software Tools for Technology Transfer*, 3(4), 2001.

[17] D. Berry, B. Wu, S. Pardon, F. Duignan, W. Grimson, P. Gaffney, F. Clarke, and J. Feely. A test request protocol system, 1999. Presentation given at the IFCC WorldLab Conference, Bologna, Italy.

[18] R. Bindels, P. A. De Clercq, R. A. G. Winkens, and A. Hasman. A test ordering system with automated reminders for primary care based on practice guidelines. *International Journal of Medical Informatics*, 58-59(1):219–233, 2000.

[19] D. Boaz, M. Balaban, and Y. Shahar. A temporal-abstraction rule language for medical databases. In *Proceeding of the workshop on Intelligent Data Analysis in Medicine and Pharmacology (IDAMAP) 2003*, 2003.

[20] D. Boaz and Y. Shahar. Idan: A distributed temporal-abstraction mediator for medical databases. In Dojat et al. [43], pages 21–30.

[21] M. H. Böhlenand C. S. Jensen. Seamless integration of time into SQL. Technical Report R-962049, Aalborg University, Department of Computer Science, Denmark, 1996.

[22] T. Bosse. An interpreter for clinical guidelines in asbru. Master's thesis, Vrije Universiteit Amsterdam, Faculty of Exact Sciences, Amsterdam, 2001.

[23] A. A. Boxwala, M. Peleg, S. W. Tu, O. Ogunyemi, Q. T. Zeng, D. Wang, V. L. Patel, R. A. Greenes, and E. H. Shortliffe. GLIF3: A representation format for sharable computer-interpretable clinical practice. *Journal of Biomedical Informatics*, 37(3):147–161, 2004.

[24] F. P. Brooks Jr. The computer scientist as toolsmith II. *Communications of the ACM*, 39:61–68, March 1996 1996.

[25] M. D. Cabana, C. S. Rand, R. Powe, A. W. Wu, M. H. Wilson, P.-A. C. Abboud, and H. R. Rubin. Why don't physicians follow clinical practice guidelines? A framework for improvement. *Journal of the American Medical Association*, 282:1458–1465, 1999.

[26] S. Chakravarty and Y. Shahar. A constraint-based specification of periodic patterns in time-oriented data. In *Proceedings of the TIME-99*, pages 29–40. IEEE Computer Society, 1999.

[27] S. Chakravarty and Y. Shahar. Specification and detection of periodic patterns in clinical data. In *Fourth Workshop on Intelligent Data Analysis in Medicine and Pharmacology (IDAMAP-99)*, pages 20–31, 1999.

[28] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed coloured nets for symmetric modelling applications. *IEEE Transactions on Computers*, 42(11):1343–1360, 1993.

[29] P. Ciccarese, E. Caffi, L. Boiocchi, A. Halevy, S. Quaglini, and A. Kumar. The NewGuide project: guidelines, information sharing and learning from exceptions. In Dojat et al. [43], pages 18–22.

[30] P. Ciccarese, E. Caffi, L. Boiocchi, S. Quaglini, and M. Stefanelli. A guideline management system. *Studies in Health Technology and Informatics*, 107(1):28–32, 2004.

[31] P. Ciccarese, E. Caffi, S. Quaglini, and M. Stefanelli. Architectures and tools for innovative health information systems: The Guide project. *International Journal of Medical Informatics*, 74(7):553–562, 2005.

[32] P. D. Clayton and G. Hripczak. Decision support in healthcare. *International Journal of Biomedical Computing*, 39:59–66, 1995.

[33] P. D. Clayton, T. A. Pryor, O. B. Wigertz, and G. Hripcsak. Issues and structures for sharing knowledge among decision-making systems: The 1989 arden homestead retreat. In L. C. Kingsland, editor, *Proceedings of the Thirteenth Annual Symposium on Computer Applications in Medical Care*, pages 116–121, New York, 1989. IEEE Computer Society Press.

[34] P. De Clercq, K. Kaiser, and A. Hasman. Computer-interpretable guideline formalisms. In ten Teije et al. [158], pages 22–43.

[35] P.A. De Clercq, J.A. Blom, A. Hasman, and H.H.M. Korsten. A strategy for development of practice guidelines for the ICU using automated knowledge acquisition techniques. *International Journal of Clinical Monitoring and Computing*, 15:109–117, 1999.

[36] P.A. De Clercq, J.A. Blom, A. Hasman, and H.H.M. Korsten. Design and implementation of a framework to support the development of clinical guidelines. *International Journal of Medical Informatics*, 64(2-3):285–318, 2001.

[37] P.A. De Clercq, A. Hasman, and B.H. Wolffenbuttel. A consumer health record for supporting the patient-centered management of chronic diseases. *Medical Informatics and Internet Medicine*, 28(2):117–127, 2003.

[38] C. Combi, E. Keravnou-Papailiou, and Y. Shahar. *Temporal Information Systems in Medicine*. Springer, 2010.

[39] P. de Clercq and A. Hasman. Experiences with the development, implementation and evaluation of automated decision support systems. *Studies in Health Technology and Informatics*, 107:1033–1037, 2004.

[40] P. A. de Clercq, J. A. Blom, H. H. Korsten, and A. Hasman. Approaches for creating computer-interpretable guidelines that facilitate decision support. *Artificial Intelligence in Medicine*, 31(1):1–27, 2004.

[41] Ed de Moel. The annotated M[UMPS] standards, 2010. Available at http://71.174.62.16/Demo/AnnoStd, last accessed October 10$^{\text{th}}$, 2010.

[42] T. Dean, L. P. Kaelbling, J. Kirman, and A. Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76(1-2):35–74, 1995.

[43] M. Dojat, E. T. Keravnou, and P. Barahona, editors. *Artificial Intelligence in Medicine, 9th Conference on Artificial Intelligence in Medicine in Europe, AIME 2003*, volume 2780 of *Lecture Notes in Computer Science*. Springer, 2003.

[44] K. Dube, E. Mansour, and B. Wu. Supporting collaboration and information sharing in computer-based clinical guideline management. In *Proceedings of 18th IEEE Symposium on Computer-based Medical Systems (CBMS 2005)*, pages 232–237. IEEE Press, 2005.

[45] J. Dufour, J. Bouvenot, P. Ambrosi, D. Fieschi, and M. Fieschi. Textual guidelines versus computable guidelines: A comparative study in the framework of the PRESGUID project in order to appreciate the impact of guideline format on physician compliance. In *Proceedings of the AMIA Symposium*, pages 219–223, 2006.

[46] J. C. Dufour, D. Fieschi, and M. Fieschi. Coupling computer-interpretable guidelines with a drug-database through a web-based system – the PRESGUID project. *BMC Medical Informatics and Decision Making*, 4(1), 2004.

[47] C. Eccher, A. Seyfang, A. Ferro, and S. Miksch. Embedding oncologic protocols into the provision of care: The Oncocure project. In *The XXII International Conference of European Federation for Medical Informatics*. IOS Press, 2009.

[48] C. Eccher, A. Seyfang, A. Ferro, and S. Miksch. Updating a protocol-based decision-support systems knowledge base: A breast cancer case study. In *Lecture Notes in Computer Science*, volume 6512, pages 126–138, 2011.

[49] M. Eccles, E. McColl, N. Steen, N. Rousseau, J. Grimshaw, D. Parkin, and I. Purves. Effect of computerised evidence based guidelines on management of

asthma and angina in adults in primary care: cluster randomised controlled trial. *British Medical Journal*, 325:941, 2002.

[50] C. Eken, U. Bilge, M. Kartal, and O. Eray. Artificial neural network, genetic algorithm, and logistic regression applications for predicting renal colic in emergency settings. *International Journal of Emergency Medicine*, 2(2):99–105, June 2009.

[51] P. J. Embi, A. Jain, J. Clark, S. Bizjack, R. Hornung, and C. M. Harris. Effect of a clinical trial alert system on physician participation in trial recruitment. *Archives of Internal Medicine*, 165(19):2272–2277, October 24 2005.

[52] K. Erol, J. Hendler, and D. S. Nau. Umcp: A sound and complete procedure for hierarchical task-network planning. In *Proceedings of the International Conference on AI Planning Systems (AIPS)*, pages 249–254, 1994.

[53] D. Fällman and A. Grönlund. Rigor and relevance remodeled. In *Proceedings of Information Systems Research in Scandinavia (IRIS25)*, 2002.

[54] M. J. Field and K. H. Lohr. *Clinical Practice Guidelines: Directions for a New Program.* National Academy Press, 1990.

[55] P. E. Friedland and Y. Iwasaki. The concept and implementaion of skeletal plans. *Journal of Automated Reasoning*, 1(2):161–208, 1985.

[56] C. Fuchsberger. Entwicklung einer Ausführungseinheit für Asbru Light. Master's thesis, Vienna University of Technology, Institute of Software Technology and Interactive Systems, 2003.

[57] A. Garg, N. Adhikari, H. McDonald, M. Rosas-Arellano, P. Devereaux, J. Beyene, J. Sam, and R. Haynes. Effects of computerized clinical decision support systems on practitioner performance and patient outcomes: a systematic review. *Journal of the American Medical Association*, 293(10):1223–1238, 2005.

[58] E. Gatziu, A. Geppert, and K. R. Dittrich. Integrating active concepts into an object-oriented database system. In *Proceedings of the 3rd International Workshop on Database Programming Languages*, pages 399–415, 1991.

[59] P. Gershkovich and R. N. Shiffman. An implementation framework for GEM encoded guidelines. In *AMIA Symposium*, pages 204–208, 2001.

[60] C. Gordon and M. Veloso. Guidelines in healthcare: the experience of the Prestige project. *Studies in Health Technology and Informatics*, 68:733–738, 1999.

[61] R. Goud, M. van Engen-Verheul, N.F. de Keizer, R. Bal, A. Hasman, I.M. Hellemans, and N. Peek. The effect of computerized decision support on barriers to guideline implementation: A qualitative study in outpatient cardiac rehabilitation. *International Journal of Medical Informatics*, 79(6):430 – 437, 2010.

[62] J. M. Grimshaw and I. T. Russell. Effect of clinical guidelines on medical practice? A systematic review of rigorous evaluations. *Lancet*, 342:317–322, 1993.

[63] A. Guarnero, M. Marzuoli, G. Molino, P. Terenziani, M. Torchio, and K. Vanni. Contextual and temporal clinical guidelines. In *Proceedings of the AMIA Symposium*, pages 683–687, 1998.

[64] G. H. Guyatt, D. L. Sackett, J. C. Sinclair, R. Hayward, D. J. Cook, R. J. Cook, and et al. Users' guide to the medical literature ix: a method for grading health care recommendations. *Journal of the American Medical Association*, 274:1800–1804, 1995.

[65] C. G. Hagerty, D. Pickens, C. Kulikowski, and F. Sonnenberg. HGML: a hypertext guideline markup language. In *Proceedings of the AMIA Symposium*, pages 325–329, 2000.

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

[67] I. J. Haimowitz, P. Phuc Le, and I. S. Kohane. Clinical monitoring using regression-based trend templates. *Artificial Intelligence in Medicine*, 7:473–496, 1995.

[68] Keith W. Hare. JCC's SQL standards page. Technical report, JCC Consulting, Inc., 2010. Webpage at http://www.jcc.com/sql.htm, last accessed October 31st, 2010.

[69] R. Haux. Medical informatics: Past, present, future. *International Journal of Medical Informatics*, 79(9):599 – 610, 2010.

[70] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1999.

[71] T. Hill and P. Lewicki. *STATISTICS Methods and Applications*. StatSoft, Tulsa, OK, USA, 2007.

[72] B. Hjorth. EEG analysis based on time domain properties. *Electroencephalography and Clinical Neurophysiology*, 29:306–310, 1970.

[73] J. Hojstrup. A statistical data screening procedure. *Measurement Science and Technology*, 4(2):153–157, 1992.

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

[75] W. Horn, S. Miksch, G. Egghart, C. Popow, and F. Paky. Effective data validation of high-frequency data: Time-point-, time-interval-, and trend-based methods. *Computers in Biology and Medicine*, 27(5):389–409, 1997.

[76] G. Hripcsak, P. Ludemann, T. A. Pryor, O. B. Wigertz, and P. D. Clayton. Rationale for the Arden syntax. *Computers and Biomedical Research*, 27:291–324, 1994.

[77] J. Hunter and N. McIntosh. Knowledge-based event detection in complex time series data. In *Artificial Intelligence in Medicine*, pages 271–280, Berlin, 1999. Springer.

[78] D.M. Van Hyfte, P.F. de Vries Robbe, T.B. Tjandra-Maga, A.A. van der Maas, and F.G. Zitman. Towards a more rational use of psychoactive substances in clinical practice. *Pharmacopsychiatry*, 34(1):13–18, 2001.

[79] D. Isern and A. Moreno. Computer-based execution of clinical guidelines: A review. *International Journal of Medical Informatics*, 77(12):787 – 808, 2008.

[80] D. Isern, D. Sanchez, and A. Moreno. HeCaSe2: a multi-agent ontology-driven guideline enactment engine. In *Proceedings of Fifth International Central and Eastern European Conference on Multi-agent Systems (CEEMAS 2007)*, volume 4696, page 322324, Berlin, 2007. Springer.

[81] P.E. Johansson, G.I. Petersson, and G.C. Nilsson. Personal digital assistant with a barcode reader–a medical decision support system for nurses in home care. *International Journal of Medical Informatics*, 79(4):232 – 242, 2010.

[82] T. R. Campion Jr., L. R. Waitman, A. K. May, A. Ozdas, N. M. Lorenzi, and C. S. Gadd. Social, organizational, and contextual characteristics of clinical decision support systems for intensive insulin therapy: A literature review and case study. *International Journal of Medical Informatics*, 79(1):31 – 43, 2010.

[83] M. G. Kahn, J. C. Ferguson, E. H. Shortliffe, and L. M. Fagan. Representation and use of temporal information in ONCOCIN. In *Proceedings of the Annual Symposium on Compututer Applications in Medical Care*, pages 172–176, 1985.

[84] H. C. Karadimas, F. Hemery, J. Simonnet, and E. Lepage. Arden/j: An architecture for MLM execution on the Java platform. *Journal of the American Medical Informatics Association*, 9:359–368, 2002.

[85] K. Kawamoto, C. Houlihan, E. Balas, and D. Lobach. Improving clinical practice using clinical decision support systems: a systematic review of trials to identify features critical to success. *British Medical Journal*, 330(7494):765, 2005.

[86] D. L. Kent, E. H. Shortliffe, R. W. Carlson, M. B. Bischoff, and C. D. Jacobs. Improvements in data collection through physician use of a computer-based chemotherapy treatment consultant. *Journal of Clinical Oncology*, 3:1409–1417, 1985.

[87] R. A. Kuhn and R. S. Reider. A C++ framework for developing Medical Logic Modules and an Arden Syntax compiler. *Computers in Biology and Medicine*, 24(5):365–370, 1994.

[88] O. Lassila and R. R. Swick. Resource description framework (RDF). model and syntax specification. w3c recommendation. Report No REC-rdf-syntax-19990222, World Wide Web Consortium, Cambridge, 1999.

[89] P. V. Le. A clinical trial of TrenDx: an automated trend-detection program. Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, 1996.

[90] E. Mansour, K. Dube, and B. Wu. Managing complex information in reactive applications using an active temporal XML database approach. In *Proceedings of the Ninth International Conference on Enterprise Information Systems (ICEIS 2007)*, pages 520–523, 2007.

[91] B. McCauley, I. Young, I. Clark, and M. Peters. Incorporation of the arden syntax within the reimplementation of a closed-loop decision support system. *Computers in Biomedical Research*, 29(6):507–518, Dec 1996.

[92] S. Miksch, W. Horn, C. Popow, and F. Paky. Utilizing temporal data abstraction for data validation and therapy planning for artificially ventilated newborn infants. *Artifial Intelligence in Medicine*, 8(6):543–576, November 1996.

[93] S. Miksch and A. Seyfang. Continual planning with time-oriented, skeletal plans. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI 2000)*, pages 511–515, Berlin, 2000. IOS Press.

[94] S. Miksch, A. Seyfang, W. Horn, and C. Popow. Abstracting steady qualitative descriptions over time from noisy, high-frequency data. In *Artificial Intelligence in Medicine*, pages 281–290, Berlin, 1999. Springer.

[95] S. Miksch, Y. Shahar, and P. Johnson. Asbru: A task- specific, intention-based, and time-oriented language for representing skeletal plans. In *7th Workshop on Knowledge Engineering: Methods & Languages (KEML-97)*, 1997.

[96] J. Nguyen, Y. Shahar, S. W. Tu, A. K. Das, and M. A. Musen. A temporal database mediator for protocol-based decision support. In *AMIA Annual Fall Symposium*, pages 298–302, 1997.

[97] M. J. O'Connor, W. E. Grosso, S. W. Tu, and M. A. Musen. RASTA: A distributed temporal abstraction system to facilitate knowledge-driven monitoring of clinical databases. In *MedInfo 2001*, pages 508–512, 2001.

[98] M. J. O'Connor, S. W. Tu, and M. A. Musen. Applying temporal joins to clinical databases. In *AMIA Annual Symposium*, pages 335–339, 1999.

[99] M. J. O'Connor, S. W. Tu, and M. A. Musen. Representation of temporal indeterminacy in clinical databases. In *AMIA Annual Symposium*, 2000.

[100] M. J. O'Connor, S. W. Tu, and M. A. Musen. The Chronus II temporal database mediator. In *AMIA Annual Symposium*, 2002.

[101] O. Ogunyemi. The Guideline Expression Language (GEL) user's guide. Technical Report DSG-TR-2000-001, Brigham and Women's Hospital, 2000.

[102] L. Ohno-Machado, J. H. Gennari, S. N. Murphy, N. L. Jain, S. W. Tu, D. E. Oliver, E. Pattison-Gordon, R. A. Greenes, E. H. Shortliffe, and O. Barnett. The

Guideline Interchange Format: a model for representing guidelines. *Journal of the American Medical Informatics Association*, 5(4):357372, 1998.

[103] M. Paesold. Monitoring temporal patterns in guideline-based care. Master's thesis, Institute of Software Technology and Interactive Systems, Vienna University of Technology, 2006.

[104] N. W. Paton and O. Diaz. Active database systems. *ACM Computing Surveys*, 31(1), March 1999.

[105] M. Peleg, S. Keren, and Y. Denekamp. Mapping computerized clinical guidelines to electronic medical records: Knowledge-Data Ontological Mapper (KDOM). *Journal of Biomedical Informatics*, 41:180–201, February 2008.

[106] M. Peleg, S. Tu, J. Bury, P. Ciccarese, J. Fox, R. Greenes, R. Hall, P Johnson, N. Jones, A. Kumar, S. Miksch, S. Quaglini, A. Seyfang, E. Shortliffe, and M. Stefanelli. Comparing computer-interpretable guideline models: A case-study approach. *Journal of the American Medical Informatics Association*, 10(1), 2003.

[107] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

[108] I. N. Purves. Clarifications and lessons from this study. In *Replies to Effect of computerised evidence based guidelines on management of asthma and angina in adults in primary care: cluster randomised controlled trial*, 2003. Available at http://www.bmj.com/content/325/7370/941/reply last accessed October 11th, 2010.

[109] S. Quaglini, M. Stefanelli, A. Cavallini, G. Micieli, C. Fassino, and C. Mossa. Guideline-based careflow systems. *Artificial Intelligence in Medicine*, 20(1):5–22, 2000.

[110] P. Ram, D. Berg, S. W. Tu, J. G. Mansfield, Q. Ye, R. Abarbanel, and N. Beard. Executing clinical practice guidelines using the SAGE execution engine. In *Proceedings Of The 11th World Congress On Medical Informatics*, pages 251–255, 2004.

[111] R. Randell and D. Dowding. Organisational influences on nurses' use of clinical decision support systems. *International Journal of Medical Informatics*, 79(6):412 – 421, 2010.

[112] J.-F. Rit. Propagating temporal constraints for scheduling. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, pages 383–388, 1986.

[113] J. Rogers, N. L. Jain, and G. M. Hayes. Evaluation of an implementation of prodigy phase two. In *AMIA Symposium*, pages 604–608, 1999.

[114] K. Rosenbrand, J. van Croonenborg, and J. Wittenberg. Guideline development. In ten Teije et al. [158], pages 1–21.

[115] S.E. Ross, L.M. Schilling, D.H. Fernald, A.J. Davidson, and D.R. West. Health information exchange in small-to-medium sized family medicine practices: Motivators, barriers, and potential facilitators of adoption. *International Journal of Medical Informatics*, 79(2):123 – 129, 2010.

[116] M. Ruzicka and V. Svatek. Mark-up based analysis of narrative guidelines with the Stepper tool. *Studies in Health Technology and Informatics*, 101:132–136, 2004.

[117] A. C. Scott, M. B. Bischoff, C. D. Jacobs, and E. H. Shortliffe. ONCOCIN: A cancer protocol consultant. In *Proceedings of the Workshop on the Role of Computers in Cancer Clinical Trials*, 1982.

[118] M. Sedlmayr. *Proaktive Assistenz zur kontextabhängigen und zielorientierten Unterstützung bei der Indikationsstellung und Anwendung von Behandlungsmanahmen in der Intensivmedizin*. PhD thesis, RWTH Aachen University, 2008. Available at http://darwin.bth.rwth-aachen.de/opus3/volltexte/2009/2688/pdf/Sedlmayr_Martin.pdf last accessed October 19$^{\text{th}}$, 2010.

[119] M. Sedlmayr, T. Rose, T. Greiser, R. Röhrig, M. Meister, and A. Michel-Backofen. Automating standard operating procedures in intensive care. In *Advanced Information Systems Engineering*, volume 4495 of *Lecture Notes in Computer Science*, pages 516–530, 2007.

[120] M. Sedlmayr, T. Rose, R. Röhrig, and M. Meister. A workflow approach towards GLIF execution. In *Workshop on AI Techniques in Healthcare - Evidence-based Guidelines and Protocols, held in conjunction with 17th European Conference on Artificial Intelligence*, page 29, 2006.

[121] B. Seroussi, J. Bouaud, E. Antoine, L. Zelek, and M. Spielmann. Using ONCODOC as a computer-based eligibility screening system to improve accrual onto breast cancer clinical trials lecture notes in computer science. In *Artificial Intelligence in Medicine*, volume 2101, pages 421–430, 2001.

[122] B. Seroussi, J. Bouaud, D. L. Denke, H. Falcoff, and J. Julien. Using knowledge modelling to measure how clinical practice could actually be evidence-based: a preliminary analysis with arterial hypertension management. *Studies in Health Technology and Informatics*, 150:668–672, 2009.

[123] B. Seroussi, J. Bouaud, H. Dreau, H. Falcoff, C. Riou, M. Joubert, C. Simon, G. Simon, and A. Venot. ASTI: a guideline-based drug-ordering system for primary care. *Studies in Health Technology and Informatics*, 84(1):528–532, 2001.

[124] A. Seyfang, K. Kaiser, and S. Miksch. Modelling clinical guidelines and protocols for the prevention of risks against patient safety. In *The XXII International Conference of European Federation for Medical Informatics*. IOS Press, 2009.

[125] A. Seyfang, R. Kosara, and S. Miksch. Asbru's reference manual, Asbru version 7.3. Technical Report Asgaard-TR-2000-3, Vienna University of Technology, Institute of Software Technology, 2002.

[126] A. Seyfang and S. Miksch. Advanced temporal data abstraction for guideline execution. In *Symposium on Computerized Guidelines and Protocols (CGP 2004)*, pages 88–102. IOS Press, 2004.

[127] A. Seyfang and S. Miksch. Asgaard's contribution to the guideline representation comparison. Technical report, Institute for Software Technology, Vienna University of Technology, Austria, 2010. Available at http://www.openclinical.org/docs/ext/cigs/comparison/Cough_documentation_Asbru.pdf, last accessed September 21$^{\text{th}}$, 2010.

[128] A. Seyfang, S. Miksch, W. Horn, M. S. Urschitz, C. Popow, and C. F. Poets. Using time-oriented data abstraction methods to optimize oxygen supply for neonates. In *Artificial Intelligence in Medicine*, pages 217–226, Berlin, 2001. Springer.

[129] A. Seyfang, S. Miksch, C. Polo-Conde, J. Wittenberg, M. Marcos, and K. Rosenbrand. MHB - a many-headed bridge between informal and formal guideline representations. In *10th Conference on Artificial Intelligence in Medicine (AIME 2005)*, pages 146–150. Springer, 2005.

[130] A. Seyfang, S. Miksch, P. Votruba, W. Reif, M. Balser, and J. Schmitt. Specification of AsbruLight. Technical Report Asgaard-TR-2004-6, Vienna University of Technology, Austria; University of Augsburg, Germany, 2004.

[131] A. Seyfang, M. Paesold, P. Votruba, and S. Miksch. Improving the execution of clinical guidelines and temporal data abstraction in high-frequency domains. In ten Teije et al. [158], pages 978–971.

[132] Y. Shahar. A framework for knowledge-based temporal abstraction. *Artificial Intelligence*, 90(1-2):79–133, 1997.

[133] Y. Shahar and C. Cheng. Knowledge-based visualization of time-oriented clinical data. In *AMIA Symposium*, pages 155–159, 1998.

[134] Y. Shahar, D. Goren-Bar, D. Boaz, and G. Tahan. Distributed, intelligent, interactive visualization and exploration of time-oriented clinical data and their abstractions. *Artificial Intelligence in Medicine*, Dec 9 2005.

[135] Y. Shahar, S. Miksch, and P. Johnson. The asgaard project: A task-specific framework for the application and critiquing of time-oriented clinical guidelines. *Artificial Intelligence in Medicine*, 14:29–51, 1998.

[136] Y. Shahar, E. Shalom, A. Mayaffit, O. Young, M. Galperin, S. B. Martins, and M. K. Goldstein. A distributed, collaborative, structuring model for a clinical-guideline digital-library. In *Proceedings of the 2003 AMIA Annual Fall Symposium*, 2003.

[137] Y. Shahar, O. Young, E. Shalom, A. Mayaffit, R. Moskovitch, A. Hessing, and M. Galperin. DeGeL: A hybrid, multiple-ontology framework for specification and retrieval of clinical guidelines. In Dojat et al. [43], pages 122 – 131.

[138] V. Shalev, G. Chodick, and A.D. Heymann. Format change of a laboratory test order form affects physician behavior. *International Journal of Medical Informatics*, 78(10):639 – 644, 2009.

[139] R. D. Shankar and M. A. Musen. Justification of automated decision-making: Medical explanation or medical argument? In *AMIA Symposium*, pages 395–399, 1999.

[140] E. H. Sherman, G. Hripcsak, J. Starren, R. A. Jenders, and P. Clayton. Using intermediate states to improve the ability of the arden syntax to implement care plans and reuse knowledge. In *Proceedings of the Annual Symposium on Computer Application in Medical Care*, pages 238–242, 1995.

[141] R. N. Shiffman, B. T. Karras, A. Agrawal, R. Chen, L. Marenco, and S. Nath. Gem: A proposal for a more comprehensive guideline document model using xml. *Journal of the American Medical Informatics Association*, 7(5):488–498, 2000.

[142] R. N. Shiffman, P. Shekelle, J. M. Overhage, J. Slutsky, J. Grimshaw, and AM. Deshpande. Standardized reporting of clinical practice guidelines; a proposal from the Conference on Guideline Standardization. *Annals of Internal Medicine*, 139:493–498, 2003.

[143] Y. Shoham. Temporal logics in AI: Semantical and ontological considerations. *Artificial Intelligence*, 33:89–104, 1987.

[144] D. Sittig, M. Krall, R. Dykstra, A. Russell, and H. Chin. A survey of factors affecting clinician acceptance of clinical decision support. *BMC Medical Informatics and Decision Making*, 6(6), 2006.

[145] S. Skonetzki, H. J. Gausepohl, M van der Haak, S. Knaebel, O. Linderkamp, and T. Wetter. HELEN, a modular framework for representing and implementing clinical practice guidelines. *Methods of Informatics in Medicine*, 43:413–426, 2004.

[146] R. T. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[147] R. T. Snodgrass. Tsql2 and sql3 interactions, 2010. Webpage at http://www.cs.arizona.edu/people/rts/sql3.html, last accessed October 31st, 2010.

[148] R. T. Snodgrass, I. Ahn, G. Ariav, D. Batory, J. Clifford, C. E. DyresonR. Z. Elmasri, F. Grandi, W. Käfer, N. Kline, K. Kulkarni, T. Y. C. Leung, N. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. M. Sripada. Tsql2 language specifcation. Technical report, Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA, September 1994.

[149] R. T. Snodgrass, M. H. Bohlen, C. S. Jensen, and A. Steiner. Transitioning temporal support in tsql2 to sql3. in , o. etzion, s. jajodia, and. In *Temporal Databases: Research and Practice*, pages 150–194, 1998.

[150] D. S. Solomon, C. J. Wroe, A. L. Rector, J. E. Rodgers, J. L. Fistein, and P. Johnson. A reference terminology for drugs. *Journal of the American Medical Informatics Association*, 1999 Special Conference Issue:152–155, 1999.

[151] J. H. Song, S. S. Venkatesh, E. A. Conant, P. H. Arger, and CM. Sehgal. Comparative analysis of logistic regression and artificial neural network for computer-aided diagnosis of breast masses. *Academic radiology*, 12(4):487–495, April 2005.

[152] M. Sordo, O. Ogunyemi, A. A. Boxwala, R. A. Greenes, and S. Tu. GELLO: An object-oriented query and expression language for clinical decision support. In *AMIA Annual Fall Symposium 2003*, page 1012, 2003.

[153] A. Spokoiny and Y. Shahar. Momentum-an active time–oriented database for intelligent abstraction, exploration and analysis of clinical data. In *Proceeding of the workshop on Intelligent Data Analysis in Medicine and Pharmacology (IDAMAP) 2003*, 2003.

[154] R. Steele and J. Fox. Enhancing conventional web content with intelligent knowledge processing. In Dojat et al. [43], pages 142–151.

[155] D. R. Sutton and J. Fox. The syntax and semantics of the PROforma guideline modelling language. *Journal of the American Medical Informatics Association*, 10(5):433–443, Sep–Oct 2003.

[156] D. R. Sutton and J. Fox. The syntax and semantics of the proforma guideline modelling language. *Journal of the American Medical Informatics Association*, 10(5):433–443, Sep 2003.

[157] V. Svatek and M. Ruzicka. Step-by-step mark-up of medical guideline documents. *International Journal of Medical Informatics*, 70(2-3):329–335, July 2003.

[158] A. ten Teije, S. Miksch, and P. Lucas, editors. *Computer-based Medical Guidelines and Protocols: A Primer and Current Trends*, volume 139 of *Studies in Health Technology and Informatics*. IOS Press, 2008.

[159] A. ten Teije, J. van Croonenborg, C. Duelli, F. van Harmelen, P. Lucas, S. Miksch, W. Reif, K. Rosenbrand, and A. Seyfang. Improving medical protocols by formal methods. *Artifical Intelligence in Medicine*, 36(3):193–209, 2006.

[160] P. Terenziani, F. Mastromonaco, G. Molino, and M. Torchio. Executing clinical guidelines: temporal issues. In *Proc AMIA Symp*, pages 848–852, 2000.

[161] P. Terenziani, G. Molino, and M. Torchio. A modular approach for representing and executing clinical guidelines. *Artificial Intelligence in Medicine*, 23(3):249–276, Nov 2001.

[162] P. Terenziani, S. Montani, A. Bottrighi, M. Torchio, G. Molino, and G. Correndo. The GLARE approach to clinical guidelines. *Studies in Health Technology and Informatics*, 101:162–166, 2004.

[163] Sven Tiffe. Defining medical concepts by linguistic variables with fuzzy Arden syntax. In *AMIA Annual Symposium*, page 796800, 2002.

[164] S. Toulmin. *The uses of argument*. Cambridge University Press, Cambridge MA, 1958.

[165] S. Tu and M. Musen. A flexible approach to guideline modeling. In *AMIA Annual Symposium*, pages 475–497, Washington D.C., 1999. Hanley & Belfus.

[166] S. W. Tu, J. G. Mansfield, T. Weida, D. Berg, K. M. Hrabak, C. Parker, J. McClay, R. McClure, M. A. Nyman, J. Glasgow, J. R. Campbell, M. A. Musen, and R. Abarbanel. The SAGE guideline model: Achievements and overview. *Journal of the American Medical Informatics Association*, 14:589–598, 2007.

[167] S. W. Tu and M. A. Musen. From guideline modeling to guideline execution: Defining guideline-based decision-support services. In *AMIA Symposium*, pages 863–867, 2000.

[168] S. W. Tu and M. A. Musen. Modeling data and knowledge in the EON guideline architecture. In *MedInfo 2001, London, UK*, 2001.

[169] M. S. Urschitz, W. Horn, A. Seyfang, A. Hallenberger, T. Herberts, S. Miksch, C. Popow, I. Mueller-Hansen, and C. F. Poets. Automatic control of the inspired oxygen fraction in preterm infants, a randomized cross-over trial. *American Journal Respiratory and Critical Care Medicine (AJRCCM)*, 170:1095–1100, 2004.

[170] F. Verhoeven, M.F. Steehouder, R.M.G. Hendrix, and J.E.W.C. van Gemert-Pijnen. Factors affecting health care workers' adoption of a website with infection control guidelines. *International Journal of Medical Informatics*, 78(10):663 – 678, 2009.

[171] R. Walters. *M Programming: A Comprehensive Guide*. Digital Press, 1997. ISBN 1-55558-167-6.

[172] D. Wang, M. Peleg, D. Bu, M. Cantor, G. Landesberg, E. Lunenfeld, S. Tu, G. E. Kaiser, G. Hripcsak, V. Patel, and E. H. Shortliffe. GESDOR - a generic execution model for sharing of computer-interpretable clinical practice guidelines. In *Proceedings of the American Medical Informatics Annual Symposium*, page 694698, 2003.

[173] D. Wang, M. Peleg, S. W. Tu, A. A. Boxwala, O. Ogunyemi, Q. T. Zeng, R. A. Greenes, V. L. Patel, and E. H. Shortliffe. Design and implementation of the GLIF3 guideline execution engine. *Journal of Biomedical Informatics*, 37(5):305–318, 2004.

[174] C. Weng, S. W. Tu, I. Sim, and R. Richesson. Formal representation of eligibility criteria: A literature review. *Journal of the American Medical Informatics Association*, 43(3):451–467, Jun 2010.

[175] B. Wu and K. Dube. PLAN: a framework and specification language with an event-condition-action (ECA) mechanism for clinical test request protocols. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, page 10, 2001.

[176] B. Wu, E. Mansour, and K. Dube. Complex information management using a framework supported by ECA rules in XML. In *Proceedings of the 2007 international conference on Advances in rule interchange and applications*, pages 224–231, 2007.

[177] J. C. Wyatt and D. J. Spiegelhalter. Field trials of medical decision-aids: potential problems and solutions. In *Proceedings of the Annual Symposium on Computer Applications in Medicine and Care*, pages 3–7, 1991.

[178] J. Van Wyk, M. van Wijk, M. Sturkenboom, M. Mosseveld, P. Moorman, and J. van der Lei. Electronic alerts versus on-demand decision support to improve dyslipidemia treatment: a cluster randomized controlled trial. *Circulation*, 117(3):371–378, 2008.

[179] O. Young and Y. Shahar. Spock: A hybrid model for runtime application of Asbru clinical guidelines. In *Proceedings Of The 11th World Congress On Medical Informatics (Medinfo) (CD)*, page 1922, 2004.

[180] O. Young and Y. Shahar. Applying hybrid-asbru clinical guidelines using the Spock system. In *AMIA Annual Symposium*, pages 854–858, 2005.

[181] O. Young, Y. Shahar, Y. Liel, E. Lunenfeld, G. Bar, E. Shalom, S. B. Martins, L. T. Vaszar, T. Marom, and MK. Goldstein. Runtime application of Hybrid–Asbru clinical guidelines. *Journal of Biomedical Informatics*, 40(5):507–526, Oct 2007.

[182] P. Yu, S. Gandhidasan, and A.A. Miller. Different usage of the same oncology information system in two hospitals in Sydney–lessons go beyond the initial introduction. *International Journal of Medical Informatics*, 79(6):422 – 429, 2010.

[183] L. Zoubek. *Automatic Classification of Human Sleep Recordings Combining Artifact Identification and Relevant Features Selection*. PhD thesis, GIPSA-lab, Universite Joseph Fourier, Grenoble, 2008.

[184] L. Zoubek, S. Charbonnier, S. Lesecq, A. Buguet, and F. Chapotot. A two-step sleep/wake stages classifier taking into account artifacts in the polysomnographic signals. In *Proceedings of the 17th World Congress of the International Federation of Automatic Control*, 2008.

[185] M. H. Zweig and G. Campbell. Receiver-operating characteristic (ROC) plots: a fundamental evaluation tool in clinical medicine. *Clinical Chemistry*, 39(4):561–577, 1993.

Curriculum Vitae

Address	Andreas Seyfang Dr. Josef Reschplatz 1/21 1170 Wien Austria
Date of Birth	May 5$^{\text{th}}$, 1966

Education

Oct. 1989 – Jun. 1996	M.S. studies in computer science at the Vienna University of Technology
Since 1996	Ph.D. studies in computer science at the Vienna University of Technology

Project Experience

Jan. 1995 – Jun. 1996	VIE-PNN: Parenteral Nutrition Solutions for Neonates. For my master thesis at the Institute of Medical Cybernetics and Artificial Intelligence, I implemented a web-based user interface which brought the break-through into practical application for an existing knowledge-based system.
Oct. 1998 – May 2002	Asgaard: Designing task-specific problem-solving methods to support the design and execution of time-oriented skeletal plans. I further developed the Asbru representation to the form currently in use, a framework to combine temporal data abstraction with plan execution, and various data abstraction algorithms, as described in this thesis.

Since 1999	Pulsoximetry: Controlling the oxigen supply in neonates based on input from pulsoximetry. I designed and implemented the data abstraction and control algorithm (in close collaboration with experts of neonatology) and actively participated in the evaluation.
Jan. 2002 – Dec. 2002 and Jan. 2004 – Jun. 2006	Protocure I + II: Integrating formal methods in the development process of medical guidelines and protocols. I designed an intermediate representation (MHB) to bridge the gap in representations between the natural language text of the guideline and Asbru. Furthermore, I modelled the sample guideline im MHB and Asbru, and designed the execution unit for Asbru.
Mar. 2007 – Dec. 2009	Oncocure: Supporting medical treatment of breast cancer by protocol execution. I created the Asbru model of the protocol used at the partner hospital and assisted the integration of the Asbru execution engine with the Electronic Patient Management System at the hospital.
Jan. 2008 – Sep. 2011	Remine: High-performance prediction, detection and monitoring platform for patient safety risk management I tought the use of MHB to partners and modelled guidelines and protocols in MHB and Asbru.
Since Nov. 2010	Brigid: Support for Authoring and Transformation of Clinical Guidelines and Protocols I develop the representations MHB and Asbru further and create a distributed, multi-author editing environment.
Publications	Please see `http://www.ifs.tuwien.ac.at/~seyfang/publications.html`