

An interpreter for clinical guidelines in Asbru

Tibor Bosse
August 2001

An interpreter for clinical guidelines in Asbru

Master thesis for the study of informatics at the Vrije Universiteit Amsterdam.

Author

Tibor Bosse
Zand-en Jaagpad 29
1396 JB Baambrugge
Student number: 0995932
E-mail address: tbosse@cs.vu.nl

Supervisors

Dr. F.A.H. van Harmelen &
Dr. M. Marcos
Department of AI
Faculty of Sciences
Vrije Universiteit Amsterdam

Project location

Vrije Universiteit
Faculty of Exact Sciences
De Boelelaan 1081-83
1081 HV Amsterdam

Project period

February – August 2001

Abstract

Clinical guidelines are seen as a promising means to improve the quality of medical care. Many knowledge-representation languages have been developed in order to represent clinical guidelines in a formal way. This document describes the implementation of an interpreter that is able to simulate the application of clinical guidelines written in the Asbru representation language. This interpreter consists of two parts, namely a Prolog parser and a Clips program. Its input is an Asbru protocol written in XML, and its output is a detailed trace of actions. Since Asbru is a very expressive language, the interpreter only supports a subset of its features, also called Asbru Light. Several test cases, for which an Asbru version of the Jaundice protocol of the American Association of Pediatrics has been used as input, have demonstrated the interpreter's capability of correctly simulating guidelines.

Samenvatting

Medische protocollen worden gezien als een veelbelovend middel om de kwaliteit van de medische zorg te verbeteren. Er zijn veel kennisrepresentatietaalen ontwikkeld om medische protocollen formeel te kunnen weergeven. Dit document beschrijft de implementatie van een interpreter die in staat is de toepassing van medische protocollen, geschreven in de representatietaal Asbru, te simuleren. Deze interpreter bestaat uit twee delen, te weten een Prolog parser en een Clips programma. Haar invoer is een Asbru protocol, geschreven in XML, en haar uitvoer is een gedetailleerde trace van acties. Aangezien Asbru een zeer expressieve taal is, ondersteunt de interpreter slechts een gedeelte van haar mogelijkheden, ook wel Asbru Light genoemd. Verschillende tests, waarbij een Asbru versie van het protocol voor Geelzucht van de American Association of Pediatrics is gebruikt als invoer, hebben de mogelijkheden van de interpreter om protocollen correct te simuleren aangetoond.

Acknowledgement

We would like to say a special word of thanks to Michael Balser, Frank van Harmelen, Robert Kosara, Mar Marcos, Sylvia Miksch, Hugo Roomans, Andreas Seyfang and Annette ten Teije for their support during our work.

Contents

1. Introduction	5
1.1 General introduction	5
1.2 Structure of this thesis	5
2. Background	6
2.1 Clinical guidelines	6
2.2 Formalisation of guidelines	7
2.2.1 Why formalisation?	7
2.2.2 Formalisation approaches	8
2.3 The Asgaard project	9
2.3.1 Origin	9
2.3.2 Current status	10
2.4 Why an interpreter is useful	11
3. Asbru in detail	12
3.1 Plan-Body	12
3.1.1 Subplans	13
3.1.2 Single-step	14
3.1.3 Cyclical-plan	14
3.2 State transitions	14
3.3 Conditions	15
3.4 Intentions	17
3.5 Preferences	17
3.6 Effects	18
3.7 Time Annotations	18
3.8 Arguments	18
3.9 Domain definitions	18
3.10 Example: the Jaundice protocol	19
4. Requirements for the interpreter	23
4.1 Behaviour	23
4.2 Asbru Light	24
4.2.1 Plan-body	24
4.2.2 Conditions and Intentions	25
4.2.3 Time Annotations	25
4.2.4 Domain definitions	26
4.3 Implementation platforms	26
5. Design choices for protocol representation	27
5.1 Representation of plans	27
5.2 Representation of expressions	29
5.3 Representation of domain definitions	31
5.4 Representation of time	32
6. The interpreter	33
6.1 The Prolog parser	33
6.1.1 Top level structure	33
6.1.2 Parsing of plans	34
6.2 The Clips program	35
6.2.1 State transitions	35
6.2.2 Evaluation of expressions	37
6.2.3 Execution of plans	39
6.2.4 Abstraction	44
6.2.5 Pre-processing	44
6.2.6 Output	45
6.3 Example output	45
6.3.1 The Prolog parser	45
6.3.2 The Clips program	47
7. Evaluation	49
8. Future work	50
Appendix A: Introduction to Prolog and Clips	51
Appendix B: Main deftemplates and defrules of the Clips program	53
Bibliography	58

1. Introduction

1.1 General introduction

At the time of the completion of this master thesis, the 21st century is already one and a half years old. A situation that still sounded unreal to many people about one decade ago. And while time is passing away very quickly, modern technology is developing at a rapid pace as well. How often one can hear people complain about 'computers taking over the world'. In other words, in many cases human beings are not happy at all with the fact that they are losing ground to automation. This is especially the case when it comes to areas that have traditionally nothing to do with computers, and where all the difficult work was always done by human specialists. A typical example for such an area is the medical world. At first sight, combining an incredibly complex, case-sensitive, knowledge-intensive job like treating diseases with the use of computers would indeed sound like science fiction. In such a field, where the smallest mistake may have the biggest consequences, people tend to have more confidence in human experts than in computer programs. And one cannot blame them for that. However, during the past decades, more and more examples of medical applications consisting of a combination of humans and computers are raising their heads. And, surprisingly or not, these are above all cases where the available knowledge is extremely numerous and complex, properties that a computer can handle particularly well.

When speaking of applications in the medical domain containing a huge amount of complex knowledge, the first things one would probably think of are clinical practice guidelines. Clinical guidelines exist in all different kinds these days, but their joint property is that they assist practitioner and patient decisions about appropriate health care for specific clinical circumstances. They were initially developed to make life easier for physicians, but have now increased so much in number and complexity, that sometimes doctors are literally '*flooded with guidelines*'[1]. And this is exactly the point where the idea of computer support comes in. A couple of years ago, an enormous wave of research was born investigating all possibilities of automated support of guidelines[2, 3, 4]. This support varies from the creation of guidelines to the verification or even the execution. And no matter what kind of objective is aimed at, when one wants to incorporate a guideline in a computer application, the information contained by the guideline has to be translated from natural language into a machine understandable format. This is why many so-called 'guideline representation languages' have been (or are still being) developed specifically for the medical domain, such as Asbru[5], GLIF[6] and PROforma[7].

One of these languages will play a prominent part in this master thesis, namely the guideline representation language Asbru. The master thesis will demonstrate an application where computers can be integrated in the domain of clinical guidelines. Note that its final goal is not to show that computers will finally replace all human specialists, or something of similar purport. (This assertion could be the subject of a whole new master thesis!). On the contrary, it will describe a small application where humans and computers will have to co-operate, being an interpreter for the guideline representation language Asbru.

1.2 Structure of this thesis

The goal of the study described in this master thesis can be defined as follows:

"Writing an interpreter for the simulation of the application of clinical guidelines, written in the Asbru language"

In other words, trying to develop a computer program that is somehow able to read in a clinical guideline, translated to Asbru, and whose output would be an exact simulation of every step that is taken when the guideline is executed for real. Other requirements for the interpreter will be defined later on in this document.

Given this goal of study, the goal of the thesis itself is simple: it will describe the exact process of the development of the interpreter, thereby giving notice of all important background knowledge, design decisions and lessons learned during this process.

Chapter 2 will give a description of the complete background of this project: what are clinical guidelines, why and how can they be formalised, what is the Asgaard/Asbru project and why is an interpreter useful for this project? Then, in chapter 3, a detailed overview of the Asbru language will be provided, with special attention for the language's most important concepts, and also an example of a real guideline written in Asbru. Chapters 4, 5 and 6 are the main parts of the document since they are dedicated to the interpreter itself. Firstly, in chapter 4, a

precise definition of the interpreter's requirements is formed, consisting of its desired behaviour, the degree to which it will support the features of Asbru, and the choice for a programming language. In chapter 5, the different design choices made for the representation of protocols are discussed. Chapter 6 will treat the real contents of the program in detail. The chapter is completed by some examples of real output. In chapter 7, the resulting behaviour of the interpreter, as well as some experiences gained during the simulation, will be judged in an evaluation. Finally, future work with respect to this project is mentioned in chapter 8.

2. Background

When taking only a short look to the title of this thesis, one can easily see that it involves three important topics, namely 'clinical guidelines', 'an interpreter' and 'Asbru'. The first topic, the notion of clinical guidelines, will be discussed below in section 2.1. This is followed by a section about the formalisation of clinical guidelines, in which several approaches are discussed. The Asbru language is one of these approaches. Since it is part of a bigger project, called Asgaard, this project is introduced in section 2.3. Finally, section 2.4. explains why an interpreter could be useful within this project.

2.1 Clinical guidelines

Obviously, when one tries to develop an interpreter for clinical guidelines, it is necessary to know exactly what clinical guidelines are. One possible definition of clinical guidelines, to which we will also stick throughout this document, is the following one:

"Clinical practice guidelines are systematically developed statements to assist practitioner and patient decisions about appropriate health care for specific clinical circumstances."[8]

Note that there exist several terms, among others 'medical guidelines', 'clinical practice guidelines' and 'medical protocols', that all mean more or less the same thing. In this document, all terms will be used indistinctly. When trying to formulate the above definition in other words, one could say that clinical guidelines prescribe how a doctor should act in certain circumstances or for certain patients. A very simplistic example might be a case where a patient suffers to complaint X, and the guideline tells the doctor to administer drug Y. However, in almost all of the cases, the structure of the guideline is many times more complex. First of all, guidelines exist in all different kinds of formats, such as narrative text, tables and flowcharts, graphs, maps, list and photographs[9]. And it is the clinician's task to interpret this information the right way, and to combine it with his or her own knowledge. Another difficulty that arises when trying to interpret a guideline, is the fact that many existing guidelines are far from complete. And what's more, sometimes they even contain incorrect information. Studies have proved that the errors that occur most are ambiguous parts, missing information, and inconsistencies[9].

Despite these difficulties, clinical guidelines have increasingly become a familiar part of clinical practice over the past decade[10]. The broad interest in clinical guidelines is already stretching across Europe (especially United Kingdom, The Netherlands, Finland, Sweden, France, Germany, Italy and Spain), North America, Australia, New Zealand and Africa. This growing interest has its origin in issues that most healthcare systems face: *"rising healthcare costs, fuelled by increased demand for care, more expensive technologies, and an ageing population; variations in service delivery among providers, hospitals, and geographical regions and the presumption that at least some of this variation stems from inappropriate care, either overuse or underuse of services; and the intrinsic desire of healthcare professionals to offer, and of patients to receive, the best care possible"*[10]. Clinicians, policy makers, and payers see guidelines as a tool for making care more consistent and efficient and for closing the gap between what clinicians do and what scientific evidence supports.

However, care should be taken to prevent that clinical guidelines are generally seen as an 'unconditional good'. They have potential benefits as well as harms[10], of which the most important ones are summarised below:

Potential benefits for patients:

- they improve health outcomes
- they improve the consistency of care, because they make it more likely that patients will be cared for in the same manner regardless of where or by whom they are treated
- they can inform the public about what their clinicians should be doing

Potential benefits for healthcare professionals:

- they offer explicit recommendations for clinicians
- their development can support quality improvement activities, because their designers have to reach agreement on how patients should be treated
- in some guideline development methods, it is usual to review existing medical studies, which makes that they can shine on gaps in the existing theories

Potential harms to patients:

- they may result in suboptimal, ineffective or harmful recommendations (e.g. because they have not been tested sufficiently)
- they may be interpreted wrongly when they are not studied accurately (e.g. a certain guideline may be best for patients overall, but inappropriate for individuals)

Potential harms to healthcare professionals:

- they may provide inaccurate scientific information and clinical advice (e.g. because they are outdated, or conflict with other guidelines)
- they can hurt clinicians professionally, when auditors and managers unfairly judge the quality of care based on criteria from invalid guidelines
- they can harm medical investigators and scientific progress if further research is inappropriately discouraged (e.g. guidelines that conclude that a treatment lacks evidence of benefit may be interpreted as grounds for not investing in further research)

Attitudes about whether clinical guidelines are good or bad for medicine vary from one group to another. Summarising, one could conclude that as long as the users succeed in avoiding their possible harms, their possible benefits appear to full advantage.

2.2 Formalisation of guidelines

2.2.1 Why formalisation?

The above section has demonstrated that clinical guidelines might bring a lot of benefits, as long as they are used with care and interpreted correctly. As we have seen already, a remarkable development that could be noticed during the last years in the domain of Artificial Intelligence is the phenomenon that many efforts were put in the formalisation of guidelines. Various representation languages have been invented, in which clinical guidelines can be rewritten in a more formal way. These new developments invoke the following question, that will be answered in this section: "what is the use of formalisation of guidelines?"

As stated before, clinical practice guidelines are seen as a tool for improving the quality and cost-efficiency of care in an increasingly complex health care delivery environment. A next step, that is often applied as well, is the automation of these formal guidelines by use of computer support. As one can imagine, computerization may enormously increase the effectiveness of both the retrieval of guidelines and the delivery of guideline-based care[3]. In an optimal scenario they would be "*integrated with the information systems operational at the point of care*". The full potentialities of computerized systems can be exploited in such an environment where different processes are executed in parallel on several patients. In this context such systems must be able to retrieve the updated situation of every patient, as well as to give an overall report on the ward, freeing the physicians to concentrate more on clinical decisions. Keeping track of the parallel activities performed, they should avoid unnecessary duplication of tasks and prevent possible omissions. Although this situation is still far away from current reality, several research projects already deal with the computer representation and implementation of guidelines[2, 11].

Nevertheless, this whole idea of translating guidelines to computer representation is a lot more difficult than it might seem at first sight. This is mainly because of the enormous gap that lies between the human-written, real-world contents of guidelines on the one hand, and the strictly formal representation that a computer needs on the other hand. And it is exactly to fill in this gap why the definition of formal models for guideline representation is required. The intended language should not only have a very clear semantics in order to avoid ambiguities, but should at the same time be very expressive as well. For instance, it should be temporally expressive and enable

designers to express complex sequential, parallel and cyclical procedures, just the way they are found in real guidelines. As an example of the latter, one could think of a treatment where the administration of a drug should be executed in parallel with a (cyclical) check for the patient's blood pressure. Moreover, when supposing this cyclical check should be performed every 10 hours, we have demonstrated an example why the language should be temporally expressive as well.

Summarising, formalisation of guidelines is a necessary intermediate step in the process of using them in computer applications, since the only guideline representation a computer can read is a formal one.

2.2.2 Formalisation approaches

The above subsection has made clear that there are several reasons why a well-defined guideline representation language is needed in this field of research. There exist many types of guideline representation languages. Besides Asbru, the object of this study (that will be explained in detail in chapter 3), we will now shortly describe two other approaches, called GLIF and PROforma, and present a comparison between the three.

Asbru

The text-based, machine-readable language Asbru is part of the Asgaard project (see section 2.3), in which task-specific problem-solving methods are developed that perform several tasks (namely design, execution and critiquing tasks, to be explained in subsection 2.3.2) in medical domains[5]. When developing this Asbru language, the researchers had themselves guided by approaches in the planning community. They have composed a list of specific tasks that often occur in plan management, and have adapted their language accordingly. The result is a language in which a guideline can be represented as a set of hierarchical skeletal *plans*. The way these plans are decomposed is extensively shown in chapter 3.

GLIF

GLIF, or the GuideLine Interchange Format, is a language designed for representing guidelines in a machine-readable format[6]. It consists of a specific data model, called the *GLIF model*. The GLIF model consists of a set of classes for guideline entities, attributes of those classes, and data-types for the attribute values. The most important object of the model is called the *GLIF guideline object*, and contains a name, a list of authors, a characterisation of the guideline's intention, a specification of the patient eligibility criteria, an unordered list of all *steps* in the guideline, an indication of the starting step in the guideline, and a list of supporting didactic material. The steps mentioned earlier can be *action steps*, specifying clinical actions that are to be performed in the patient-care process, and *decision steps*, directing the flow between guideline steps. All the steps, including the *starting step*, contain a pointer to their successor step(s), thereby defining the sequence of decisions and actions. Some of the decision steps contain *criteria*, which are conditions that can be evaluated in order to define the flow from one step to another.

PROforma

PROforma is a methodology for decision support systems, and in particular for clinical procedures in the medical domain[7]. It is intended for supporting the complete process from early knowledge acquisition to the construction of an executable system. PROforma contains a semi-formal language for describing medical decision-making procedures, a tool for constructing knowledge-level models of such procedures, and a tool for executing such models. Building a clinical protocol in PROforma is done in two phases. In the first phase, a high level description of the protocol is made with a graphical editor. In the second phase the graphical design is instantiated with knowledge needed for the enactment of the protocol. The *task* is the top level structure in PROforma, all the other structures are derived from it. PROforma supports four basic classes of tasks:

- *plan*: a sequence of several sub-tasks or components. Plan components usually have an ordering to specify temporal, logical or source constraints.
- *decision*: is used where a candidate must be chosen from a given set using arguments pro and contra.
- *action*: a procedure that has to be executed outside of the computer system, like an injection.
- *enquiry*: requests information needed to execute a certain procedure.

Every task has some specific attributes, e.g. pre- and postconditions for a plan task.

Comparison

In [9], these three representation languages have been compared, from which the following conclusions can be drawn:

- the background and purpose of all three languages are quite similar
- Asbru has the most extensive possibilities for knowledge modelling, but a strong aspect of GLIF is the possibility of adding didactics and supplemental material to the model
- PROforma and Asbru use a hierarchical structure for the representation of sequences of actions, where GLIF uses pointers. Although both techniques could lead to a correct guideline representation, the GLIF pointer structure does not allow the reuse of step(s) in other guideline representations
- PROforma and Asbru have the most extensive possibilities for making transitions between events, because precise criteria can be defined determining when certain actions are appropriate
- GLIF and PROforma divide the guideline modelling process in two phases; in Asbru there is no intermediate model

As can be deduced from these conclusions, all three languages have their pros and cons. However, it is notable that Asbru has the most extensive possibilities for both knowledge modelling and making transitions between events. For instance, it contains structures to represent sequential, parallel and cyclical procedures, just like they occur in real guidelines. It is especially this expressiveness that justifies the choice of using Asbru as the subject for this master thesis, since it makes that Asbru covers nearly all requirements of this domain.

2.3 The Asgaard project

The above section has demonstrated why Asbru was a good choice as the subject for this master thesis. However, before being able to dig into the different elements of this complex language, something should be said about the project it is part of, the Asgaard project.

2.3.1 Origin

The need to improve the quality of health care has led to a strong demand for clinical protocols and computer systems supporting both their creation and execution. This demand was the basis for the birth of the Asgaard project[12]. As stated before, the participants in this project were initially inspired by the planning community. However, old approaches in the planning community concentrated especially on algorithmic improvements, but mostly failed in medical applications. The most relevant characteristics of the medical domain are the following:

- large domain knowledge is available, but is often uncertain and vague
- incomplete and non-deterministic information about the world state and effects of actions
- unobservable underlying processes determine the observable world states
- world states, events, actions, plans, goals, and effects are durative and with uncertainty in their occurrences
- use of multiple time lines based on different granularities by providing reference annotations
- procedures can be suspended

Characteristics that are important, but hold in other domains too, are:

- a huge volume of data
- pre- and postconditions are needed to control the execution of durative actions or plans
- a goal may not be achievable in time
- sequential, parallel, and cyclical execution of procedures is necessary

Thus, a guideline representation language was needed that would be able to handle these characteristics. On the other hand, it had to incorporate the strong points of the planning community as well. The most urgent characteristics needed in an appropriate plan representation language are:

- hierarchical decomposition of plans
- knowledge rich control structure
- temporal dimension

- possibility to express the context, the general circumstance in which an action takes place

The Asgaard/Asbru project tried to build the bridge between planning approaches and the medical approaches, addressing the demands of the medical staff on the one side and applying rich plan management on the other side. The designers have developed a time-oriented, intention-based, skeletal plan-specification language, called *Asbru*, specific to the set of plan-management tasks. 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 be flexibly expressed by bounding intervals
- all plans or some plans might be executed in sequence, parallel, in a particular order, or periodically
- explicit intentions and preferences can be stated for each plan separately

In the Asbru language, a protocol can be represented by a hierarchy of plans. A plan itself consists of a name, a set of arguments, including a *time annotation* specifying the duration of the plan's execution, and the following five components (also called *knowledge roles*), which are explained in more detail in chapter 3:

- *preferences* bias or constrain the selection of a plan to achieve a given goal
- *intentions* are high-level goals at various levels of the plan, represented as temporal patterns of provider actions and patient states, at different levels of abstraction, that should be maintained, achieved, or avoided
- *conditions* are temporal patterns, sampled at a specified frequency, that need to hold at particular plan steps to induce a particular state transition of the plan instance
- *effects* describe the functional relationship between either (1) each of the relevant plan arguments and measurable parameters it affects in certain contexts, or (2) the overall plan and the clinical parameters it is expected to effect
- the *plan-body* is a set of plans to be executed in parallel, in sequence, in any order, or in some frequency

Apart from the language itself, several tools have been developed (or are still under construction) to facilitate working with Asbru. Examples are a user interface that allows users to enter specific data about a treatment[13], a data abstraction unit designed for the abstraction from raw data to abstract concepts used in therapeutic plans[14], and several design supporting tools[15]. Some of these tools will be treated below.

2.3.2 Current status

As for the current status of the project: the first evaluation results in the language design branch were very encouraging, in the sense that the proposed solution truly meets the demands of the medical staff and its concept is comprehensible and lucid to computer scientists too. Several people with different backgrounds, who were not involved in the developing process of the project, have written the same guidelines of general practitioners in Asbru independently. As it turned out, they all came up with roughly the same structure of protocols[12].

A next intended task of the project is the critiquing task. In the field of critiquing, a problem specification and a solution proposed by the user are given as input, and a series of comments aimed at improving the solution is generated as output[16]. One can imagine that this can be very useful in the medical domain, because in that domain there are often many possible solutions. In the traditional view on critiquing, standards are used to give feedback on the user's behaviour. A rather innovating way of critiquing, for which Asbru has actually been used, is to turn the tables and use the user's solution and its justification to refine the standards[16]. This form of critiquing is done in three steps:

- 1) modelling a protocol using a specific protocol language (Asbru)
- 2) manually critiquing the solutions given by an expert to a set of predefined cases by comparing them against the prescriptions of the protocol. In order to do this, not only the actions made by the expert are compared with the protocol, but also his or her intentions.
- 3) drawing conclusions about the utility of critiquing for protocol improvement

Yet, the first conclusions drawn from this field of research were slightly disappointing. It turns out that this form of critiquing is a very difficult task, especially because mostly there is no 1:1 correspondence between expert's actions and parts of the protocol, but rather a variety of relationships. In addition to this, the direct use of intentions for critiquing is hardly possible either. This is because the intentions of a protocol are not stated explicitly, which makes them very hard to model, and the intentions reported by an expert almost always differ.

However, these results do not mean that the research with respect to critiquing has stopped. On the contrary, it is still very interesting to see if critiquing can contribute to the improvement of the quality of medical protocols. Thus, future research in this field of science will try to solve the mapping problem between the expert and the protocol, e.g. by using additional knowledge to fill in the gaps.

The third task the Asgaard project was meant for, that of execution, has had the least attention as yet. This is partly explicable by the fact that actual computerized execution of medical guidelines is simply an extremely difficult thing to achieve. It is so difficult because the output of such an executor would depend on an enormous amount of (non-deterministic) parameters. Nevertheless, at the moment designers have made a start with a computational module focusing on real-time execution[11]. This module will be compatible with the data abstraction unit mentioned earlier[14]. The function of this data abstraction unit is to translate the raw data that it receives as input from the real world to more abstract concepts as they occur in protocols. As soon as the abstraction has been done, the abstract concepts can be used by the execution unit.

A last task the Asgaard project is concerned with, that has not been mentioned earlier, is the verification of clinical guidelines. By this we do not mean the verification of protocols by means of critiquing techniques, but rather a more formal verification, to make sure that the guidelines in question are correct. There have already been verification studies for Asbru. In one of them, where the verification was limited to checks for completeness and unambiguity, the protocols were examined up to three levels of a plan: the plan itself, all its knowledge roles and all its subplans[17]. In order to make the language complete and unambiguous, the following specifications had to hold at the three levels of the plan hierarchy:

- every condition must have a chance to be satisfied
- every valid sequence of plan states must be reachable
- every plan must be able to complete
- a subplan should not be stopped by its parent plan
- no state of a plan should be skipped
- no plan must loop eternally

Another, more formal verification approach of guidelines in Asbru is done at the university of Augsburg[18]. Here, the Asbru protocols are translated to a program in the KIV language, which is then used for the purpose of theorem proving. Compared to the verification approach described above, we could describe the KIV one as dynamic verification, while the completeness checks were static. The KIV project is still active at the moment.

2.4 Why an interpreter is useful

Sections 2.1 to 2.3 have demonstrated that the Asgaard project has already achieved several successes. Researchers have managed to translate some guidelines of medium size to Asbru, and critiquing as well as verification techniques have been applied to some of these guidelines. However, one thing that is still missing until now is an interpreter that would be able to execute or simulate the application of the guidelines by means of a computer. As mentioned before, a computational module focusing on real-time execution is under development. The intended users of such an executor will be real medical staff, and it will have a nice user interface, to be able to operate in the real world. Yet, it will probably take some time before this piece of software is fully operational. This is the point where another option shows up, namely the construction of a small interpreter that is much less complex, but that will also be much easier to implement. And this is what became the subject of this thesis: an interpreter, not in the first place designed for medical staff, but above all for Asbru protocol designers. With this interpreter, the intended users would be able to simulate the execution of their own created Asbru protocols while designing them. This would allow them to compare the output of the interpreter with the intended behaviour of their Asbru protocols, so that they can check the correctness of these protocols at any desired moment in the designing process.

3. Asbru in detail

The aim of this chapter is to give a rather detailed description of the most important concepts of version 7.2 of the Asbru language, on which the interpreter is based. It will in particular deal with the concepts that play an important part in the interpreter, so that the next chapters will be easier to understand. However, this description is not meant to be complete. A complete description of Asbru 7.2 can be found in the Asbru 7.2 reference manual[19]. A more concise introduction to Asbru concepts can be found in [20].

A medical protocol is modelled in Asbru as a library of hierarchical skeletal plans. Skeletal plans are "*plan schemata at various levels of detail, that capture the essence of the procedure, but leave room for execution-time flexibility in the achievement of particular goals*"[21]. When translating a protocol into Asbru, each different procedure in the protocol will be represented as an Asbru plan. Examples could be general plans to perform a whole diagnosis or treatment, but also smaller subplans in which a certain value (e.g. of blood-pressure) is observed. The idea of the hierarchical structure is that, when the guideline would actually be executed, parent plans can activate their children, which in their turn activate their children. The leaves of this hierarchical tree are atomic plans consisting of just a simple interaction with the clinician or an external device. All plans in Asbru are identified by a unique name, and can be composed of a time annotation, preferences, intentions, conditions, effects, and a plan-body, all optionally. An example of a plan is shown in figure 1. Note that this plan is completely fictive. All of its values have been made up, in order to give a good representation of the extensive possibilities of Asbru plans. Moreover, the format that the plan is presented in has been made up as well, because this notation is visually more attractive than the XML-syntax, in which Asbru protocols should in reality be modelled. The exact meaning of all elements will become clear when reading this chapter.

PLAN	Plan-1
TIME ANNOTATION	([_,_], [_,24 hours], [_,_], *NOW*)
PREFERENCES	Select-method: exact-fit
INTENTIONS	Avoid intermediate state: (glucose-level = high)
CONDITIONS	Abort-condition: (glucose-level = high) Filter-condition: ((patient-age > 60) AND (patient-age < 80))
EFFECTS	Plan-effect: Parameter="glucose-level" Relationship="decrement" Likelihood: 0.65
PLAN-BODY	Parallel subplans: Continuation specification: (treatment-1 OR treatment-2) Diagnosis Treatment-1 Treatment-2

Figure 1. Example of a fictive Asbru plan. The time annotation and the preference state that the plan should be finished within exactly 24 hours. According to the intentions and conditions, it should avoid that the level of glucose becomes high, and if so, it should abort. It should only be applied to patients between 60 and 80 years old. Its effect is a decrement of the level of glucose, with a probability of 65%. The plan-body contains three subplans to be executed in parallel, namely Diagnosis, Treatment-1 and Treatment-2.

We will now explain all different elements of Asbru in different sections. Section 3.1 to 3.8 will respectively treat the concepts of plan-body, state transitions, conditions, intentions, preferences, effects, time annotations, and arguments. Except for the state transitions, these can all be (possible) elements of an Asbru plan. Section 3.9 will explain the notion of domain definitions. Finally, an example of an existing Asbru protocol will be given in section 3.10.

3.1 Plan-Body

The plan-body is the part of the plan in which the biggest part of its behaviour is defined, because it specifies the dynamics of the plan. If the plan is decomposed into sub-procedures, the plan-body specifies these sub-procedures, as well as the way they are ordered. However, in some cases a plan-body cannot be decomposed into smaller parts. Altogether, it can have one of the following forms, according to the Asbru 7.2 reference manual:

- *subplans*: a set of plan steps performed e.g. in parallel or sequentially
- *cyclical-plan*: a plan repeated several times

- *single-step*: a single step of plan execution
- *for-each-plan*: a plan performing a certain action for each element in a list or set
- *iterative-plan*: a plan performing a loop of single plan steps
- *refer-to*: a reference to the plan-body of another plan
- *to-be-defined*: a special tag declaring that this plan is not executable
- *user-performed*: a special tag showing that this plan is executed through some action by the user, which is not further modelled in the system

We will now discuss the first three of these elements in more detail, since they occur often in practice and require further explanation.

3.1.1 Subplans

In those cases when the plan-body consists of several smaller parts, the element subplans is used. It is used to group subplans (which can be single steps as explained in subsection 3.1.2, or other plans) in one of the following temporal orderings:

- *Sequential*: the steps (or subplans) are performed in strict order. Each is started after its predecessor is finished.
- *Parallel*: all steps are started at the same time. They may finish at any time.
- *Any-order*: the steps are performed one at a time, without strict ordering. For example, the second step can be performed before the first one, but at each instant in time, only one step is performed, i.e. their execution must not overlap.
- *Unordered*: there is no ordering between the steps. They may overlap or not, and each may start and end whenever appropriate. Still, like in the other three cases, each of them starts after the start of the parent plan and the parent finishes when the last one of the children finishes.

Besides the ordering of the subplans, the element *subplans* must be provided with a certain waiting strategy. This waiting strategy consists of the elements *continuation specification*, *wait-for-optional-subplans*, and *retry-aborted-subplans*, explained below.

Continuation specification

The so-called *continuation specification* is a logical expression, based on the names of the subplans, which must be fulfilled before the parent plan can complete successfully. For example, consider the continuation specification of plan-1 in figure 1. This says that plan-1 only completes when either the subplan “treatment-1” or the subplan “treatment-2” have completed. Furthermore, there exist three special symbols, namely *all*, *one* and *none*, indicating that respectively all, one or no subplans must complete before successful completion of the parent. Whenever a subplan must complete in order to satisfy a plan’s continuation specification, we say that this subplan is *mandatory*. Otherwise, it is *optional*.

Wait-for-optional-subplans

In addition to the continuation specification, each plan has an attribute called *wait-for-optional-subplans*. It is *no* by default which means that the plan terminates as soon as both complete condition (to be explained in section 3.3) and continuation specification evaluate to true. If *wait-for-optional-subplans* is set to *yes*, the parent waits for its children to complete as long as the time annotation (section 3.7) of its activation allows, even if they are not necessitated by the continuation specification.

Retry-aborted-subplans

The normal behaviour of most plans is to try each subplan once. Still in some cases, one wants to try a set of subplans over and over again, until one or more of them succeed. For such cases, the attribute *retry-aborted-subplans* can be set to *yes*. The time annotation and the *abort condition* (explained in section 3.3) of such plans should ensure termination in cases where children permanently fail.

3.1.2 Single-step

With the abstract element *single-step* actions that cannot be decomposed into smaller actions (e.g. prompting for the input of the patient's age) can be represented. The element stands for one of the following:

- *plan-activation*: activates another plan
- *variable-assignment*: assigns a value given by an arithmetic expression to a variable
- *set-context*: sets a context-variable to particular value
- *ask*: asks the user to enter a new value for a certain parameter
- *list-manipulations*: performs a manipulation to a list (e.g. insert an element)
- *if-then-else*: if the condition given evaluates to true, the first single step is taken, otherwise the second is performed

In addition to this list, the single step *plan-activation* may be provided with a special entity, namely the so-called *on-abort* plan. This is an alternative plan, that should be executed when the activated plan fails (i.e. is aborted). Imagine for instance the situation that a disease can be cured by two different treatments, and one of them does not work. In that case, the other treatment is tried as an alternative.

3.1.3 Cyclical-plan

Apart from the different temporal structures that can be made with the possibilities provided in subsection 3.1.1, there exists another situation that cannot be modelled with these means. And this is the situation where a single plan has to be repeated several times within a certain time interval. That this situation is realistic can simply be illustrated by thinking of a drug that should be taken every morning, or a test that should be performed once a week. To be able to represent this kind of information, the *cyclical-plan* has been included in Asbru. It is a complex element that consists of the following parts:

- *start-time* evaluates to a time point, at which the action specified by *cyclical-plan-body* is performed for the first time.
- *cyclical-plan-body* is a single plan step, mostly a *plan-activation*. This step is repeated as specified in the repeat specification.
- *any-repeat-specification* specifies the times at which the step defined by *cyclical-plan-body* is repeated.
- *set-of-cyclical-complete-conditions* is a combination of cyclical complete conditions (e.g. a certain time point has been reached, or the step has been repeated several times), connected by Boolean operators.
- *max-attempts* is the number of attempts made to complete the action given in *cyclical-plan-body*, before the plan aborts.

An example of a cyclical plan-body is shown in figure 2. This plan states that the blood pressure of the patient should be asked every 12 hours. The plan should start immediately and should end whenever the blood pressure has been asked successfully 5 times. However, if it fails for 3 times in succession, the plan should abort.

Cyclical:	
Start-time:	*NOW*
Cyclical-plan-body:	Ask blood-pressure
Any-repeat-specification:	every 12 hours
Set-of-cyclical-complete-conditions:	5 times-completed
Max-attempts	3

Figure 2. Example of a cyclical plan-body. The cyclical operation is to ask for the patient's blood pressure.

3.2 State transitions

State transitions are a very important aspect of the Asbru language. They are based on the notion of *plan states*. The idea is that every plan in the library must, at any time during the execution, be in one of the following states: considered, possible, ready, rejected, activated, suspended, aborted, and completed. During the execution, a plan may switch between states, just like this is done in the real medical world (e.g., a certain treatment may at first be suspended, and be reactivated later in better circumstances). The choices whenever a plan may switch between states are based on the notion of conditions (to be explained in section 3.3). A complete state-transition diagram

is shown in figure 3. As can be deduced from this figure, all plans start in state *considered*. After this, they normally switch to state *possible* and *ready*, as long as they are not *rejected*. Then, they are *activated*, and if all goes well, they are *completed*. However, from state *activated*, they may switch to state *aborted* or *suspended* (and eventually back to *activated*) as well.

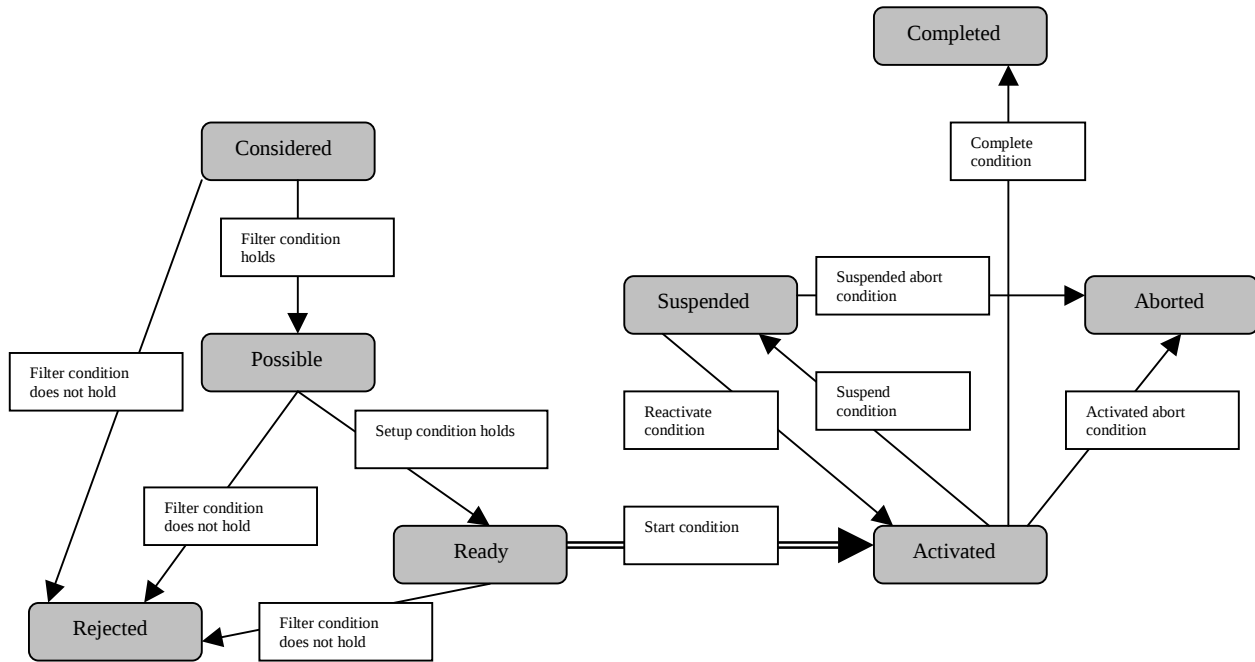


Figure 3. Asbru state-transition diagram. The rounded rectangles represent the different states an Asbru plan may have.

Besides the transitions shown in this diagram, there are other reasons why a plan may switch to another state, namely because of *propagation*. Propagation occurs whenever a parent plan must switch to another state because of a transition of one of its children. For instance, imagine a plan that has a certain subplan within its continuation specification (which means that the plan can only complete if the subplan does). Whenever this subplan aborts, the parent plan will never have a chance to complete anymore, so it is aborted as well. In that case we say that the abortion is propagated from the child to the parent.

3.3 Conditions

Conditions are very powerful structures within Asbru, since they influence the applicability of plans, among others. They need to hold at particular plan steps to induce a particular state transition of the plan. This is why conditions can be seen as state-transition criteria that specify the transition between neighbouring plan states. They are represented as logical expressions of many different kinds, combined by logical operators. In addition, they can have temporal patterns. Seven types of conditions are distinguished:

- *Filter precondition*: must hold when the plan is considered and cannot be achieved by alternative means (e.g. the patient is a pregnant female). If the filter precondition is fulfilled, the plan becomes *possible*. If it does not hold, the plan is *rejected* (see figure 3).
- *Setup precondition*: must hold before the plan can be executed. It can be achieved through actions by the physician (e.g. the patient had a glucose-tolerance test). If the setup precondition is fulfilled, the plan becomes *ready*. Otherwise, it can be fulfilled by activating suitable plans. If the setup precondition is not fulfilled during a predefined time period (specified by means of a time annotation) then the plan is *rejected*.
- *Activate mode*: is always set to either *default* or *automatic*. If the activate mode is *automatic*, the plan is started (i.e. it becomes *activated*) immediately after becomes *ready*. If it is *manual*, the user is asked for approval before the plan is started.

- *Suspend condition*: determines that an active plan instance must be *suspended*, as soon as it is fulfilled (e.g. blood glucose has been high for four days). Should the suspend condition become false later, the plan still stays suspended. Only if the reactivate condition is fulfilled, the plan can be reactivated.
- *Reactivate condition*: determines when a suspended plan has to be *activated* again (e.g. blood glucose level is back to normal or is only slightly high).
- *Complete condition*: determines when a plan is *completed* (i.e. it finishes successfully; e.g. delivery has been performed). As long as the complete condition is not fulfilled, the plan stays active (if it is not aborted or suspended).
- *Abort condition*: determines when an active or suspended plan has to be *aborted* (e.g. there is an insulin-indicator condition: the patient cannot be controlled by diet). If this plan is a mandatory subplan of the plan which activated it, that plan will also be aborted by propagation.

In addition, a condition may contain an *explanation slot*. This is an important place reserved for the explanation of the condition. For instance, whenever a certain plan aborts because of a critical condition, it is often necessary to know the exact reason for this abortion. The explanation slot allows the guideline modeller to clear up this reason.

Furthermore, whenever a plan has several subplans, the conditions of these subplans are not always checked in the same order. This is because the way the state transitions of the different subplans are synchronized depends on the type of the parent plan (see subsection 3.1.1), as can be seen in the following explanation:

- *Sequential*: the second plan's filter condition is not checked before the first plan completes or aborts (see figure 4).
- *Parallel*: first, all filter conditions are checked. Only after the last one becomes fulfilled, all setup conditions are checked. After all setup conditions are fulfilled, the activate mode is checked. If a single mandatory subplan is rejected, the parent plan is aborted and no other subplan is started (see figure 5).
- *Any-order*: the preselection phase (checking of filter and setup condition and activate mode) is done in parallel, without synchronization. The first plan which passes this phase is executed. Other plans remain idle after being activated, so that only one plan at a time remains activated. As soon as the plan executed first terminates (successful or not) or is suspended, the plan which finished its preselection phase second, is executed (see figure 6).
- *Unordered*: all phases are executed (potentially) in parallel without any synchronisation (see figure 7).

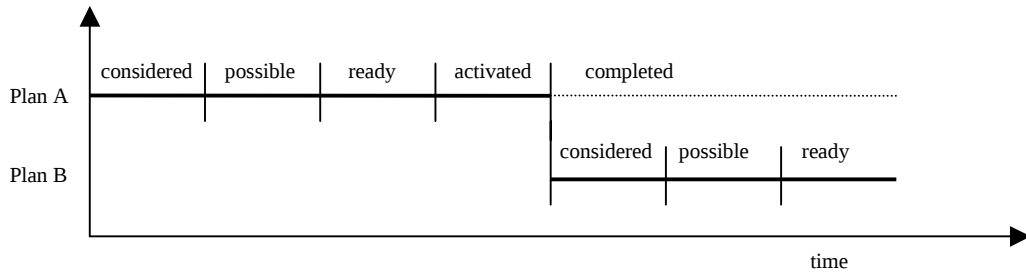


Figure 4. Timing of sequential plans.

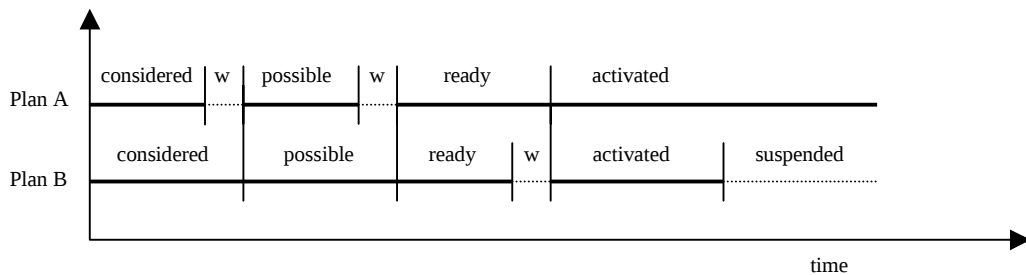


Figure 5. Timing of parallel plans.

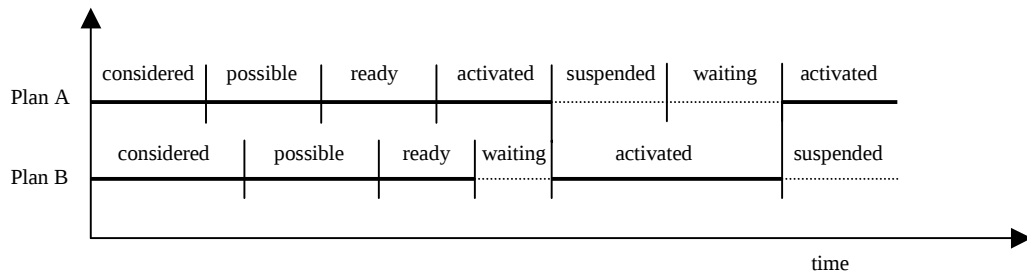


Figure 6. Timing of any-order plans.

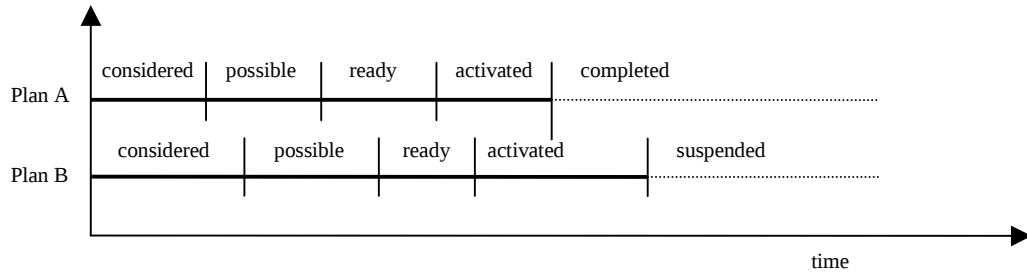


Figure 7. Timing of unordered plans.

3.4 Intentions

When performing actions with respect to medical diagnosis or treatment, it is often useful to realise why this action is being performed after all. Hence, in Asbru too, some mechanism would be needed to express the aim of the plan. This would be especially useful to support special tasks as critiquing, as we have seen before. And that is why the notion of intention was created. Intentions are high-level goals at various levels of the plan, represented as temporal patterns of clinician actions and patient states, that should be maintained, achieved, or avoided. On the one hand, there are states and actions which should be true during the execution of a plan. On the other hand, there are states and actions which should be true after the execution of a plan. Altogether, four categories of intentions are defined:

- *Intermediate state*: the patient state(s) that should be maintained, achieved or avoided during the applicability of the plan (e.g. weight gain levels should be slightly low to slightly high)
- *Intermediate action*: the clinician action(s) whose execution should be maintained, achieved or avoided during the execution of the plan (e.g. monitor blood glucose once a day)
- *Overall state pattern*: the overall pattern of patient states that should have been maintained, achieved or avoided after finishing the plan (e.g. patient had less than one high glucose value per week)
- *Overall action pattern*: the overall pattern of clinician actions whose execution should have been maintained, achieved or avoided after finishing the plan (e.g. patient had visited dietician regularly for at least three months)

3.5 Preferences

Preferences bias or constrain the selection of a plan to achieve a given goal. Thus, when at any moment of the execution a choice has to be made between multiple subplans, preferences can play a role in this selection. Examples of preferences are[5]:

- *strategy*: a general strategy for dealing with the problem (e.g. an aggressive or normal treatment)
- *utility*: a set of utility measures (e.g. minimize the cost or the patience inconvenience)
- *select-method*: a matching heuristic for the applicability of the whole plan (e.g. exact-fit, which means that certain time annotations (section 3.7) must hold exactly, not approximately)
- *resources*: a specification of prohibited or obligatory resources (e.g. in certain cases of treatment of a pulmonary infection, surgery is prohibited and antibiotics must be used)

3.6 Effects

Effects describe the expected behaviour of the plan's execution, independent of the complete condition and the intentions. An effect is specified by one of the following constructs:

- *Argument-dependency*: describes the functional relationship between each of the relevant plan arguments and the measurable parameters it affects in certain contexts. For example, an effect may state that the dose of insulin has a negative-monotonic relationship to the level of blood glucose of the patient.
- *Plan-effect*: describes the relationship between the overall plan and the clinical parameters it is expected to effect. For instance, an insulin-administration plan would decrease the blood-glucose level.

As in the case of preferences, effects have been created in order to make it easier to select plans. For instance, whenever there is a choice between two different plans, one can take a look at the effects to see whether one of them has an undesirable effect. In addition to what has been explained earlier, all effects can have a likelihood annotation, specifying the probability of occurrence.

3.7 Time Annotations

Since time aspects have a crucial role in many clinical guidelines, Asbru had to provide a mechanism by which these aspects could be expressed. Examples of temporal aspects are the duration of plans, or the period during which a certain condition should hold. Nevertheless, one problem is the fact that most temporal aspects found in guidelines have a high degree of uncertainty. Imagine for instance a plan that has to be performed between now and two weeks, or a plan of which it is impossible to predict exactly when it will finish. This is why the time annotations in Asbru allow a representation of uncertainty in starting time, ending time, and duration of a time interval. This way, a time annotation is written as follows:

([ESS, LSS], [EFS, LFS], [MinDu, MaxDu], REFERENCE)

With this notation, the temporal annotation can indicate for each plan its earliest starting shift (ESS), latest starting shift (LSS), earliest finishing shift (EFS), latest finishing shift (LFS), minimal duration (MinDu), maximal duration (MaxDu), and a reference point. As an example, take a look at the following time annotation, meaning that the plan "starts 24 to 26 weeks after the conception, ends 32 to 34 weeks after the conception, and lasts 7 to 9 weeks":

([24 WEEKS, 26 WEEKS], [32 WEEKS, 34 WEEKS], [7 WEEKS, 9 WEEKS], CONCEPTION)

Note that all of these values can be unknown (denoted by an "_") to allow incomplete time annotations. The special symbol *NOW* can be used as a shortcut for the current time (as for instance in figure 1, where the plan must finish within 24 hours from the current time). Also note that, whilst the above examples concerns the duration of a plan, time annotations occur most in combination with conditions.

3.8 Arguments

Besides the six main parts just treated, an Asbru plan can be provided with a set of *arguments*. Arguments exist in two kinds, namely input and output arguments, which are nothing more than referenced values that can be passed to another plan, in a way similar to programming languages. An example is a simple plan that adds a certain value to each element of a list. The input arguments of that plan could for instance be the initial list and the particular value, and the output argument could be the resulting list, so that another plan could use these results.

3.9 Domain definitions

When a guideline is being executed, many different kinds of world information are involved (e.g. the patient's age, the patient's blood pressure, the colour of the patient's urine, the possibility of a cholestatic disease, and so on). Within Asbru, there exist several kinds of data that are used to store this information. The ensemble of these data structures is called *domain definitions*. We will now treat some of them in more detail.

Variables

A variable is a value whose temporal dimension is ignored. It simply evaluates to its current value, like variables in a programming language do. There are local and global variables, depending on if they are declared for a single plan or for the whole plan library.

Parameters

As opposed to variables, parameters are observed over time. Their values may for instance be calculated based on a reference to another parameter, or on a logical combination of Boolean parameters. Parameters can also be used for something that is called *abstraction*. When applying an abstraction some quantitative data is being turned into qualitative data. For example, when the patient's age is between 0 and 24 hours, the qualitative parameter 'age' is set to 'day1'.

Constants

Constants are used in the same manner as they are in programming languages. Their values are assigned within the Asbru code of the protocol, and do never change. For example, the constant 'max_dose' may have the value '1000'.

As an additional remark, notice that domain definitions are completely independent from the Asbru plans and are thus stored at another place in the XML file, or even in a separate file.

3.10 Example: the Jaundice protocol

In order to illustrate the aspects of Asbru treated in previous sections more clearly, we will in this section discuss (part of) the contents of an existing Asbru protocol. The document in question describes the Asbru model of a Hyperbilirubinemia - or Jaundice- protocol[22], constructed by scientists from the Vrije Universiteit Amsterdam. This Asbru protocol was based on the guideline for the 'Management of Hyperbilirubinemia in the Healthy Term Newborn' from the American Academy of Pediatrics[23]. The domain of this guideline is neonatal medicine. It contains recommendations from specialists based on a comprehensive literature search and data analysis, and it is addressed to non-specialist health-care professionals. The guideline is divided into an evaluation part (largely diagnostic), a treatment part and an appendix. The format of the original version was a combination of narrative text, lists, tables and flowcharts. Besides giving a nice illustration of different Asbru features, this protocol is also important because it will form the main test case for the interpreter, discussed in chapter 5.

The top level plan of the Jaundice protocol is called *Hyperbilirubinemia*, and is shown in figure 8, again in an informal syntax. In this plan, checks for rapid TSB increase, for jaundice after 2 weeks and after 3 weeks run in parallel with the diagnosis and treatment parts, included together in the *Diagnostics&Treatment-hyperbilirubinemia* plan. The only requirement for the successful completion of the current plan is the completion of Diagnostics&Treatment-hyperbilirubinemia, as stated in the continuation specification. On the other hand, the plan should abort as soon as Check-for-rapid-TSB-increase activates, uncovering the possibility of a hemolytic disease.

PLAN	Hyperbilirubinemia
TIME ANNOTATION	
PREFERENCES	
INTENTIONS	Avoid intermediate state: (bilirubin = transfusion)
CONDITIONS	Abort-condition: (possibility-of-hemolytic-disease = yes)
EFFECTS	
PLAN-BODY	Unordered subplans: <i>Wait for optional subplans</i> <i>Continuation specification:</i> (Diagnostics&Treatment-hyperbilirubinemia) Check-for-rapid-TSB-increase Check-for-jaundice>2-weeks Check-for-jaundice>3-weeks Diagnostics&Treatment-hyperbilirubinemia

Figure 8. Plan Hyperbilirubinemia. Top level plan of Jaundice protocol.

The plans *Check-for-rapid-TSB-increase*, *Check-for-jaundice>2-weeks* and *Check-for-jaundice>3-weeks* are relatively simple plans. They consists mostly of ask and if-then-else statements. For illustration we will only show one of them in detail, namely *Check-for-jaundice>2-weeks* (see figure 9). A remarkable construction inside this plan is the fact that the filter condition contains a time annotation. This time annotation states that the plan only becomes active when jaundice is still present after 2 weeks (see the figure, both the latest starting shift and the earliest finishing shift have been set to “2 weeks”) after the patient’s date of birth. Furthermore, this plans demonstrates the possibility of having nested constructs, for instance an any-order and if-then-else construct inside a sequential plan.

PLAN	Check-for-jaundice>2-weeks
TIME ANNOTATION	
PREFERENCES	
INTENTIONS	Achieve overall state: (known(possibility-of-cholestatic-disease))
CONDITIONS	Filter-condition: (jaundice-clinically-significant = yes), ([_,2 weeks], [2 weeks, _], [_,_], Birth-Date)
EFFECTS	
PLAN-BODY	Sequential subplans: Continuation specification: all Any-order subplans: Continuation specification: all Ask physical-exam-OK Ask colour-stools Ask colour-urine If ((physical-exam-OK = no) OR (colour-stools = light) OR (colour-urine = dark)) Then Ask direct-serum-bilirubin Possibility-of-cholestatic-disease = yes Exit-possibility-of-cholestatic-disease Else Possibility-of-cholestatic-disease = no Exit-provide-routine-care

Figure 9. Plan Check-for-jaundice>2-weeks.

As opposed to the plans that perform only checks, the plan *Diagnostics&Treatment-hyperbilirubinemia* (see figure 10) is very important since it groups the two main procedures of the guideline, the diagnosis and treatment parts. This plan can be considered as completed when the diagnosis part deems that jaundice is not significant or when the treatment part completes successfully. The plan should abort when applied either to non-term or one-day babies. Besides, it should also abort as soon as the plan *Diagnostics-hyperbilirubinemia* aborts, uncovering a pathologic reason.

The diagnosis plan *Diagnostics-Hyperbilirubinemia* (figure 11) performs actions looking for a possible pathologic reason and investigating whether jaundice is significant, in this order. In principle it is considered that no pathologic reasons exist. The plan only completes when all actions have been completed. The abort of this plan can be caused by the abort of any of the necessary actions that it includes, because at any point, an underlying pathologic reason can be discovered.

PLAN	Diagnostics&Treatment-Hyperbilirubinemia
TIME ANNOTATION	
PREFERENCES	
INTENTIONS	Avoid intermediate state: (bilirubin = transfusion)
CONDITIONS	Abort-condition: ((term-child = no) Explanation: "Exiting the protocol to individualized clinical evaluation, including assessment of jaundice in light of prematurity" OR (age = day1) Explanation: "Exiting the protocol to individualized clinical evaluation, including assessment of jaundice and non-immune hemolytic disease" OR (pathologic-reason = yes)) Complete-condition: ((jaundice-clinically-significant = no) Explanation: "Follow this infant in routine clinical supervision" OR (completed(treatment-hyperbilirubinemia)))
EFFECTS	
PLAN-BODY	Sequential subplans: <i>Continuation specification:</i> none Ask Term-child Ask Age-child Diagnostics-hyperbilirubinemia Treatment-hyperbilirubinemia

Figure 10. Plan Diagnostics&Treatment-Hyperbilirubinemia.

PLAN	Diagnostics-Hyperbilirubinemia
TIME ANNOTATION	
PREFERENCES	
INTENTIONS	Achieve overall state: ((known(pathologic-reason)) AND (known(jaundice-clinically-significant)))
CONDITIONS	
EFFECTS	
PLAN-BODY	Sequential subplans: <i>Continuation specification:</i> all Pathologic-reason = no Anamnesis-abnormal-signs Blood-tests Anamnesis-hemolytic-disease Jaundice-determination

Figure 11. Plan Diagnostics-Hyperbilirubinemia.

The treatment plan *Treatment-Hyperbilirubinemia* (figure 12) consists of two parallel parts, namely the actual treatments and a cyclical ask for the input of new age and TSB values. Regarding the treatments, either the *Regular-treatments* or *Exchange-transfusion* can take place depending on the bilirubin level. Besides, exchange transfusion should be applied in case of failure of the regular treatments. Note that this possibility of executing an alternative plan in case of a failure is modelled with the statement *on-abort*. The treatment block completes when either Regular-treatments or Exchange-transfusion do it, as stated in the continuation specification.

PLAN	Treatment-Hyperbilirubinemia
TIME ANNOTATION	
PREFERENCES	
INTENTIONS	Avoid intermediate state: (bilirubin = transfusion) Achieve overall state: (bilirubin = observation)
CONDITIONS	
EFFECTS	
PLAN-BODY	Parallel subplans: <i>Continuation specification: all</i> Any-order subplans: <i>Continuation specification: one</i> Regular-treatments On-abort Exchange-transfusion Exchange-transfusion Cyclical: Every 12-24h Ask Age-Child, TSB-value

Figure 12. Plan Treatment-Hyperbilirubinemia.

We could of course continue by showing the rest of the plans of the Jaundice protocol, but think that the given selection satisfies for a good comprehension of its structure. On the other hand, one thing that is interesting is to show an Asbru plan in the syntax for which they were initially designed, namely XML. Figure 13 shows the Hyperbilirubinemia plan, written in XML.

```

<plan name="Hyperbilirubinemia">
  <intentions>
    <intention type="intermediate-state" verb="avoid">
      <simple-condition>
        <comparison type="equal">
          <left-hand-side>
            <parameter-ref name="bilirubin"/>
          </left-hand-side>
          <right-hand-side>
            <constant-ref name="transfusion"/>
          </right-hand-side>
        </comparison>
      </simple-condition>
    </intention>
  </intentions>
  <conditions>
    <abort-condition>
      <simple-condition>
        <comparison type="equal">
          <left-hand-side>
            <parameter-ref name="possibility-of-hemolytic-disease"/>
          </left-hand-side>
          <right-hand-side>
            <constant-ref name="yes"/>
          </right-hand-side>
        </comparison>
      </simple-condition>
    </abort-condition>
  </conditions>
  <plan-body>
    <subplans type="unordered" wait-for-optional-subplans="yes">
      <wait-for>
        <static-plan-pointer plan-name="Diagnostics-and-Treatment-hyperbilirubinemia"/>
      </wait-for>
      <plan-activation>
        <plan-schema name="Check-for-rapid-TSB-increase"/>
      </plan-activation>
      <plan-activation>
        <plan-schema name="Check-for-jaundice-after-2-weeks"/>
      </plan-activation>
      <plan-activation>
        <plan-schema name="Check-for-jaundice-after-3-weeks"/>
      </plan-activation>
      <plan-activation>
        <plan-schema name="Diagnostics-and-Treatment-hyperbilirubinemia"/>
      </plan-activation>
    </subplans>
  </plan-body>
</plan>

```

Figure 13. Plan Hyperbilirubinemia, written in XML.

4. Requirements for the interpreter

Before being able to make a start with the implementation of an interpreter for Asbru protocols, its most important requirements should be defined. These requirements involve all different kinds of aspects, of which the most important ones are the behaviour of the interpreter (among others what input and output it will have), the degree to what it will support the features of Asbru, and the choice for a programming language. These three aspects will be discussed respectively in sections 4.1 to 4.3.

4.1 Behaviour

As mentioned in section 2.4, the intended users of the interpreter are in the first place Asbru plan designers. As stated there, the interpreter would support them while translating guidelines to Asbru, because of the possibility of detecting errors in the behaviour of the protocol they just wrote. Besides, it could be used to simulate already existing guidelines and see if they produce the desired behaviour.

Another important aspect to consider when building the interpreter is what the input it receives will look like. Given its main goals mentioned above, it is obvious that the input will be a complete Asbru protocol. However, there are several options for the format this protocol is presented in. It would be nice if they could just be left in the informal representation as shown in figure 1, since this representation is visually very attractive. Unfortunately, this is impossible for the simple reason that these figures have not been modelled according to the Asbru syntax. A more suitable solution is to use the notation as shown in figure 13, the more formal XML syntax. This way, the problem about the Asbru syntax is solved. However, this approach would still bring two drawbacks, namely:

- (1) before being able to interpret a guideline, it should be written in XML first, which can be very time-consuming.
- (2) since the XML representation of protocols is declarative, the interpreter will have to make a rather complex translation to interpret this representation as a set of procedural instructions.

In fact, none of these drawbacks are exclusive to XML. The first one, the problem of the translation to XML, is an inevitable consequence of modelling a guideline in a formal way. Choosing another syntax would not change this problem. On the other hand, there is a possibility to solve the second drawback. As stated in subsection 2.3.2, researchers are working on the translation of Asbru guidelines to the KIV language, for the purpose of theorem proving. KIV is a language whose syntax is much closer to that of a programming language than XML, so using Asbru protocols translated to KIV as input might be an option. Nevertheless, at the moment this work has not advanced far enough to ensure that the KIV protocols reflect the same semantics as the XML protocols, since this simply has not been proved yet. Moreover, the translation process has not been automated, but should be done by hand, which increases the risk of producing incorrect translations. This is why the final choice for the subject of this thesis became to build an interpreter that takes (part of) an Asbru guideline, written in XML, as input.

The output of the interpreter will have the form of a trace of actions that are done during the execution, part of which will be user interactions (e.g. requests for new world information, or displays of explanation slots). The more detailed the steps in this trace are, the more useful for the designer, at least to a certain extent. Just mentioning that a plan is started is not very interesting. It would be much more useful if every single step of the state transition diagram of figure 3 was printed for each plan. The next thing that had to be decided is the format in which these output steps will be presented. Despite a set of interesting options, we have finally decided to leave the output in plain text. A rationale for this choice will be given in a later section, since it has been made during, and not before, the implementation process.

Now that the input and output of the program have been fixed, it is necessary to choose some test cases, which will help to judge its behaviour when it is finished. The choice for these test cases became versions of the *Jaundice Protocol* (introduced in section 3.10) and a *Diabetes Protocol*[24]. There already existed an informal Asbru version of both protocols, so they needed to be rewritten in XML in order to be useful test input. These two protocols were chosen because they are of medium size, but make use of almost all of the important aspects of Asbru. Hence, they represent an example of the minimal requirements the interpreter must be able to handle when it is finished.

Some other decisions made concerning the planned behaviour of the interpreter are the fact that it will not have the disposal of a nicely elaborated user interface, nor it will show real-time behaviour. Given the intended user group of the software, namely Asbru plan designers, the first choice is logical because they will not have an urgent need for a visually nice interface, as long as the program output is understandable. Using much efforts for the user interface will therefore be an unnecessary waste of time. As for the real-time behaviour: having this would mean that the execution of a guideline would be simulated second for second. It is obvious that this is exactly the desired behaviour when the program would operate in a real medical environment, but in the current context it might not be desirable. To what extent time steps will be modelled then, will also be explained in a later section.

4.2 Asbru Light

Writing a computer program that supports an extremely expressive language like Asbru is a difficult task. Making it capable of supporting all the features that Asbru has would take an enormous amount of time. Moreover, the risk that a small part of it has errors in it would be big. That is why the choice was made to pick a selection of the complete set of features of Asbru, and to design the interpreter for the time being just for this selection. This way, the risk of not seeing the wood for the trees is reduced. The chosen selection mentioned above was made during working sessions of researchers concerned to the Asgaard project, and is called *Asbru Light*. However, the development of the interpreter was not the only thing that made the researchers define this smaller version of Asbru. In fact, the selection was also a result of several practical cases, among which the modelling of the Jaundice and Diabetes protocol. It turned out that these protocols made use of only part of the elements that Asbru supports. Hence, in certain situations (e.g. when modelling these protocols) it would be nice to have a language that provides only the necessary features. This language became Asbru Light. It will be described in this section, as well as an explanation why we made this particular selection. The first subsection will discuss the plan-body, the second will discuss conditions and intentions, the third will discuss time annotations, and the last subsection will discuss domain definitions. Note that preferences and effects have not become part of Asbru Light. The explanation for this is that they hardly ever occurred within the original Jaundice and Diabetes protocols. And since modelling them would mean a lot of extra work without any important gain, at least for this application, it was decided to keep them out of Asbru Light.

4.2.1 Plan-body

Subplans

As said before, the Jaundice and Diabetes protocol make use of a lot of the important control structures that Asbru provides. And indeed, they contain all of the four types of plans mentioned in subsection 3.1.1, being *sequential*, *parallel*, *any-order* and *unordered* plans. As a consequence, all types have become part of Asbru Light. Another thing that became part of Asbru Light is the entire idea of the waiting strategy. This concerns among others the *continuation specification*, because this is also widely used in the test protocols, and is in fact an essential part of many average-sized guidelines. Next to the continuation specification, the attributes *wait-for-optional-subplans* and *retry-aborted-subplans* have been left in Asbru Light too, for the same reasons.

Cyclical-plan

Although they do not really occur often in the Jaundice protocol, it was finally decided to allow the possibility of making cyclical plans in Asbru Light too. This is because sometimes a cyclical plan can be important, and often there is no other way to model this behaviour. An existing example, drawn from the Diabetes protocol, is a cyclical plan that increases the dose of a certain drug every 2 to 4 weeks. However, the interpreter's possibilities of executing different types of cyclical plans will probably be guided by the specific cases in the test protocols. It will therefore not be capable of supporting the complete cyclical plan as defined in the reference manual (subsection 3.1.3). For instance, a simple cyclical *ask* starting 'now' will be needed, but not all possible combinations of complex starting and finishing shifts will be implemented.

Single-step

Since only a subset of the types of single plan steps, as described in subsection 3.1.2, occur in the test protocols, it has been decided to keep this subset in Asbru Light. The single plan steps meant are the *ask*, *variable-assignment*, *plan-activation*, and *if-then-else*. Note that the most frequent of these steps is the *if-then-else*

statement, which makes it possible to do a condition-based action within the plan-body. Furthermore, the decision of allowing *ask* and *variable-assignment* steps entails that different data structures should be supported by the interpreter. A way should be found to deal with simple data, stored in variables and parameters, but also with more complex data like lists. Lastly, the *on-abort plans* have been included in Asbru Light as well, since they are an essential part of the *plan-activation*.

4.2.2 Conditions and Intentions

Conditions

Both the Jaundice and the Diabetes protocol contain *Filter Preconditions*, *Activate Modes*, *Complete Conditions* and *Abort Conditions*. They do not contain any other types of conditions, so that the choice with respect to Asbru Light was simple: only the conditions just mentioned are in it. Note that an important consequence of this choice is that the only possible states a plan in Asbru Light can have are *considered*, *ready*, *rejected*, *activated*, *aborted* and *completed*. Without the *Setup Precondition*, *Suspend Condition* and *Reactivate Condition*, the states *possible* and *suspended* are not reachable anymore. In other words, the Asbru Light version of the state transition diagram would be identical to figure 3, after the removal of these two states. This also means that the state transitions would be slightly different. However, we decided that the notion of propagation remains the same.

As for the atomic expressions in which conditions can be decomposed, only a subset of them occurs in the test protocols, and is thus included in Asbru Light. This subset only contains the following types of atomic conditions:

- (x equal y)
- (x not-equal y)
- (x greater-than y)
- (x less-than y)
- (x completed)

Intentions

Intentions are rather often present in the current versions of the Jaundice and Diabetes protocol. As a matter of fact, they were not there initially, but have been acquired by the modellers because they needed them for the purpose of critiquing, so that they are present now. However, for the moment, intentions are no parts of Asbru that have any influence on the execution of the guideline. They have been created as a handy extra feature for the guideline designer, to help him or her realise what is the goal of a certain plan. But no matter if they are left out or not, the execution remains the same. That is why all types of intentions could be considered as being part of Asbru Light, but in the current practice they will only be parsed by the interpreter, without doing anything else with it. Adapting the interpreter in such a way that it can do something with the intentions remains future work.

4.2.3 Time Annotations

As stated in a previous section, implementing a program with exact real-time behaviour might not be desirable. However, not modelling any notion of time at all would go a bit too far, especially because it is sometimes used in both the Jaundice and Diabetes protocol, the test cases for this project. The way time is used there is in fact only in conditions, and of course in case of cyclical plans. An example of a condition containing a time annotation is the following filter condition, meaning that the plan becomes ready when jaundice is still present 2 weeks after the baby's birth:

(jaundice-clinically-significant = yes), ([_,2 weeks], [2 weeks, _], [_,_], Birth-Date)

Since this type of condition is not exceptional in the protocols in question, it has been decided to make simple time annotations in conditions part of Asbru Light. Nevertheless, besides time annotations in conditions, Asbru also supports time annotations that constrain the duration of a plan (see section 3.7). We have decided to leave the latter kind out of Asbru Light.

4.2.4 Domain definitions

A quick scan of the Jaundice and Diabetes protocol indicates that they contain variables, parameters as well as constants. Since the need for all three is rather obvious, they are kept for Asbru Light. The same holds for the notion of abstraction, since this will also be necessary to model the test protocols. For instance, the Jaundice protocol makes use of the qualitative parameters ‘bilirubin’, which represents the bilirubin level in relation with recommended therapy, and ‘TSB-change’, which represents the magnitude of the change in TSB-value between the last two measurements.

4.3 Implementation platforms

Now that the exact behaviour of the interpreter has been defined, a choice for an appropriate programming language has to be made. This again is not a very obvious task. In fact, we are not dealing with a problem for which only one specific language is appropriate. Three possible options, the languages *Prolog*, *Clips*, and *Jess*, have been investigated more closely, for reasons that follow.

The program has to parse its input one way or the other. Since this input is presented in XML style, it would be a very long-winded process to write this complete parser by hand. Fortunately, there exist many languages with XML parsers. For instance, there is a *Prolog* version that has a special package, designed for XML-parsing[25]. In addition, the program has to do pattern matching on the XML structure. Prolog is very good at this kind of pattern matching, which makes this a very promising option. A possible drawback of Prolog is that it is especially designed for search problems, which our project is not. It has for instance a very powerful backtracking mechanism, that will probably not be needed at all.

Clips[26] is a language that uses a set of facts that hold in a certain state of the world, together with a set of rules that can fire in certain situations. If it were possible to represent the different actions that occur in Asbru protocols as these rules in Clips, this would be a very good option as well. A particularly nice aspect of Clips is the fact the behaviour of a Clips engine is already similar to the intended behaviour of the interpreter, namely continually checking conditions to see which ones have become true, and then execute the corresponding action.

Lastly, the possibility of using a modern object oriented language like *C++* or *Java* has not been considered, because of personal preferences of the programmer. However, there was one alternative language, called *Jess*. Jess is a language that is based on Clips, but that works in Java. Its major advantage is that it has much more features for debugging. However, all possibly usable version of Jess seemed to be still under construction, which made this option considerably less attractive.

After weighing up all pros and cons, the final choice became to split up the program into two different parts, of which one will be written in Prolog and the other one in Clips. The general idea is that the Prolog part reads in as Asbru protocol written in XML, parses this by means of the XML-parser package, and produces thereupon a file of Clips facts as output. Subsequently, this file of Clips facts (which will consist of several plans written in a chosen Clips representation) will serve as input for the second part of the interpreter, the Clips program. It is this program that will do the real work, namely simulating the execution of the guideline in question, and thereby producing a trace as final output. This approach might seem a bit of a long way round, but its big advantage is the fact that it combines the strong points of the two programming languages. Given this approach, the whole series of steps from a medical protocol to the output of our interpreter would look like this:

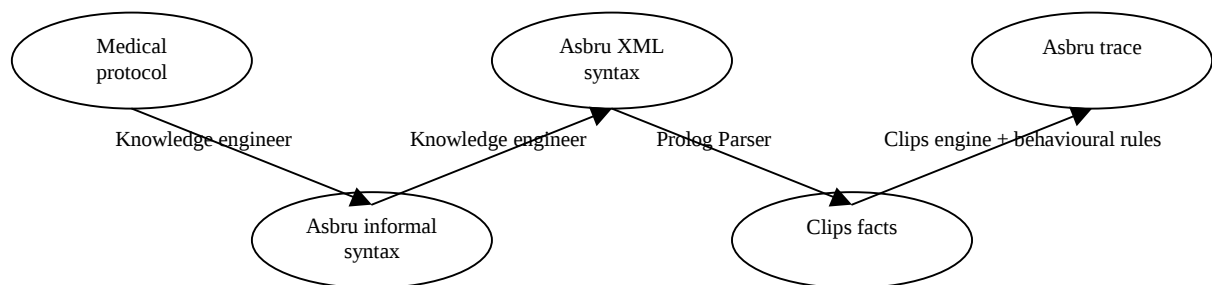


Figure 14. Series of steps from a medical protocol to an Asbru trace.

5. Design choices for protocol representation

Writing an executor for a complex language like Asbru is obviously not a straightforward process. At many moments during the design process, important decisions concerning the translation from the Asbru plans to the Clips language had to be taken. The following sections respectively describe the way we represent plans, expressions, variables, and other elements. Note that they only demonstrate how the (static) facts can be created. The way these facts are controlled by means of rules will be explained in the next chapter.

5.1 Representation of plans

The most important concept in the Asbru language is that of a *plan*. Hence, the first design choice for the interpreter was how to represent a plan in Clips. In order to do this, we decided to simply take a look at the (Asbru Light) plans in the Jaundice and Diabetes protocol, and figure out which attributes would be needed to represent these plans. It turned out that often the most complicated part of the plans is the plan-body, and that this is also the only knowledge role that is used in all the plans, so we chose to especially focus on the plan-body. A study of the characteristics of all plan-bodies led to an extensive template for the representation of plans in Clips, which can be found in Appendix B. As an illustration, figure 15 provides part of this template, containing only four basic attributes. A brief introduction to Clips is provided by Appendix A.

(deftemplate plan

```
(slot name
  (type SYMBOL))
(slot type
  (type SYMBOL)
  (allowed-symbols do_unordered do_sequentially do_any_order do_parallel cyclical if_then_else
    ask assignment manipulation display user_performed do_for_each))
(slot subplans
  (type SYMBOL))
(slot state
  (type SYMBOL)
  (allowed-symbols ignored considered possible ready activated aborted completed))
...
)
```

Figure 15. Partial template for the representation of a plan in Clips.

As can be seen in the figure, the minimal attributes a plan needs are a *name*, a *type*, a list of *subplans* and a *state*. Note that the plan state *ignored* has been added to the list of allowed states, meaning “not considered yet”. This was done to make a distinction between the state of plans before and after they have been considered (i.e. when their parents have been activated). Thus, the ignored state is nothing more than an initial value for a plan that has not even been considered. Given this representation, an example of a simple plan could look like this:

```
(plan (name plan1) (type do_sequentially) (subplans plan2 plan3) (state ignored))
```

The plan-body of this plan, called 'plan1', states that the subplans 'plan2' and 'plan3' should be executed in sequence. This way, all kinds of control structures could be used in the slot 'type', making it possible to build every desirable tree of Asbru plans. The leaves of these trees would then be plans that do not have any subplans, like asks and variable-assignments. However, some problems would arise with this representation when a plan-body contains nested statements. That is why we made the constraint that every plan has exactly one type. In case of nested statements, we decided to divide the plan into different subplans that all have only one type. For instance, plan *Check-for-jaundice>2-weeks* of the Jaundice protocol (in figure 9) would be divided into the following three plans:

```
(plan (name check-for-jaundice>2-weeks) (type do_sequentially)
  (subplans check-for-jaundice>2-weeks_1 check-for-jaundice>2-weeks_2) (state ignored))
(plan (name check-for-jaundice>2-weeks_1) (type do_any_order) (subplans ...) (state ignored))
```

(plan (name check-for-jaundice>2-weeks_2) (type if_then_else) (subplans ...) (state ignored))

Note that, although they are more or less represented the same way, there is one big difference between the original plans, to which we will also refer as *proper plans* from now on, and the new plans that will be created while parsing, to which we will refer as *fictive plans*. This difference is the fact that fictive plans cannot have any conditions, which makes that they can only be in state ignored, activated, aborted or completed. We are very well aware of the fact that this decomposition does not correspond exactly to the original meaning of the Asbru plan. On the other hand, from a programmer's point of view, it makes life much easier because it enormously decreases the complexity of single plans. Hence, the effect will be that the number of plans is increased, but their complexity will be reduced. Obviously, it remains to be seen whether this difference will lead to much difficulties of understanding for the intended users of the resulting interpreter.

Besides, as can be read in the allowed symbols of the *type* slot, we have decided to represent all single plan steps as plans too. This choice makes that, besides the distinction 'proper plans-fictive plans', we can make another distinction. This is the distinction between the single plan steps, which we will call *atomic* plans from now on, and the other, *non-atomic* plans, no matter if they are proper plans or fictive plans. A schema showing all four possible combinations is provided in figure 16. Here, the plans in question are printed in boldface. As can be seen in the schema, a plan whose body consists of just an *ask* (or another single plan step) is an atomic proper plan. If a plan has a body with subplans (i.e. it is composed), then the plan itself is a non-atomic proper plan. If these subplans are single plan steps, they are atomic fictive plans. If not, they are non-atomic fictive plans.

	Atomic	Non-atomic
Proper plan	<u>Ask value1</u>	<u>Parallel subplans</u> Ask value1 Ask value2
Fictive plan	Parallel subplans <u>Ask value1</u> Ask value2	Parallel subplans <u>Parallel subplans</u> Ask value1 Ask value2 Ask value3

Figure 16. Difference between atomic, non-atomic, proper plan, fictive plan.

The template provided in figure 15 was of course far from complete. Since a plan can have much more attributes than was sketched there, many additional slots had to be defined to be able to store them. As said before, the complete template defined for modelling plans is included in Appendix B. Altogether, the attributes can be divided into five classes, namely those that are mandatory for all plans, those that are only mandatory for the non-atomic plans, those that are optional but will be set to a default value by the program when they are not specified, those that are optional without a default value, and some internal attributes that are only interesting for the programmer and not for the user. Note that by stating that an attribute is mandatory, we mean that the Clips fact defining that plan must contain that attribute before running the program. The exact meaning of the attributes of all classes is explained below.

Mandatory attributes for all plans

- *name*: a symbol defining the name of the plan
- *type*: a symbol representing either a plan with subplans, a cyclical plan or a single plan step
- *proper*: either *yes* or *no*, depending on if the plan is a proper plan

Mandatory attributes for non-atomic plans

- *subplans*: a list containing the names of the plan's subplans, if any
- *cont_spec*: a symbol defining the name of the continuation specification, whose contents are defined elsewhere (to be explained in section 5.2)

Optional attributes with default value

- *state*: a symbol defining the state of the plan; it has *ignored* as default value

- *retry_aborted*: either *yes* or *no*, depending on the value of *retry-aborted-subplans*; default is *no*
- *wait_optional*: either *yes* or *no*, depending on the value of *wait-for-optional-subplans*; default is *no*
- *activation*: either *automatic* or *manual*, depending on the value of *activate mode*; default is *automatic*

Optional attributes without default value

- *filter_cond*: a symbol defining the name of the filter condition, whose contents are defined elsewhere (see section 5.2). Note that we assert these names to the conditions at the moment that we parse them.
- *abort_cond*: a symbol defining the name of the abort condition
- *complete_cond*: a symbol defining the name of the complete condition
- *input_args*: a list of all input arguments the plan has
- *output_args*: a list of all output arguments the plan has
- *repeat*: in case the plan is a cyclical plan, this slot contains the numbers and units of the corresponding repeat specification

Internal attributes

- *invoked_by*: a list containing all *invokers* the plan has (to be explained in subsection 6.2.3)
- *your_turn*: either *yes* or *no*, depending on if the plan is allowed to perform a step (to be explained in subsection 6.2.3)
- *since*: a number containing the point of time when the plan was activated

Note that these attributes will be understood better when reading section 6.2 about the Clips program. Some example facts representing real plans from the Jaundice protocol are the following:

```
(plan (name Hyperbilirubinemia) (proper yes) (abort_cond abort_Hyperbilirubinemia) (type
do_unordered) (wait_optional yes) (cont_spec Diagnostics-and-Treatment-hyperbilirubinemia)
(subplans Check-for-rapid-TSB-increase Check-for-jaundice-after-2-weeks Check-for-jaundice-after-3-
weeks Diagnostics-and-Treatment-hyperbilirubinemia))
```

```
(plan (name Check-for-jaundice-after-2-weeks) (proper yes) (filter_cond filter_Check-for-jaundice-
after-2-weeks) (type do_sequentially) (cont_spec all) (subplans Check-for-jaundice-after-2-weeks_1
Check-for-jaundice-after-2-weeks_2))
```

```
(plan (name Exit-possibility-of-cholestatic-jaundice) (proper yes) (type user_performed))
```

Notice that the type of the non-atomic plan Hyperbilirubinemia is ‘unordered’, the type of the non-atomic plan Check-for-jaundice-after-2-weeks is ‘sequential’, and the type of the atomic plan Exit-possibility-of-cholestatic-jaundice is ‘user-performed’. All plans are proper plans.

An important remark is that the facts we have just shown are also used for execution purposes. In other words, we do not make a distinction between the representation of plans before, during or after the execution. The only difference is that before the execution, all plans have state *ignored*. The consequences of this decision will be discussed later on, when evaluating the interpreter’s behaviour.

5.2 Representation of expressions

Besides the plan, another object that can be found frequently within Asbru Light is the *logical expression*. Expressions can be seen as logical combinations of parameter values, connected by the logical operators *and*, *or*, and *not*. In Asbru, the value of an expression can be *true*, *false*, or *unknown*, when it cannot be determined. With respect to the interpreter, it is of course important to confirm which aspects have to be represented with logical expressions. These turned out to be the following three concepts:

- 1) conditions
- 2) continuation specifications
- 3) if-branches in if-then-else statements

In (1) and (3), the atomic parameters combined by the logical operators are expressions evaluating a certain variable (e.g. *Abort-condition: ((term-child = no) OR (age = day1) OR (pathologic-reason = yes))*), whilst in

(2), the parameters combined by the logical operators are simply plan names (e.g. *Continuation specification: (subplan-1 AND (subplan-3 OR subplan-4))*). Note that the expressions can be nested as well. In case a continuation specification is defined by the special symbol *all*, this could be represented by combining a logical *and* with the names of all subplans. Similar translations could be made for the symbols *one* and *none*. Although there exist thus two different kinds of expressions, we decided to model them both by means of one single template, and to distinguish between the two by means of a special slot, named *type*. Furthermore, the choice was made to translate one nested expression to several simple expressions, exactly the same way as with nested plans were dealt with (see section 5.1). The consequence of this decision is that each expression must have exactly one logical operator. All this resulted in the template shown in figure 17.

(deftemplate exp

```

  (slot name
    (type SYMBOL))
  (slot label
    (type SYMBOL))
  (slot refer_to
    (type SYMBOL))
  (slot type
    (type SYMBOL)
    (allowed-symbols spec cond opt))
  (slot operator
    (type SYMBOL)
    (allowed-symbols not or and atomic))
  (multislot subexps)
  (multislot period)
  (slot since)
  (slot explanation)
  (multislot value
    (type SYMBOL)
    (allowed-symbols true false unknown)
    (default unknown))
)
```

Figure 17. Final template for the representation of an expression in Clips.

The attributes involved in this template can be divided into three classes, namely mandatory, optional and internal attributes. Their meanings are discussed below.

Mandatory attributes

- *name*: a symbol defining the name of the expression; slots that are part of plans (e.g. *filter_cond*, *cont_spec*) can refer to this name
- *label*: a symbol defining the label of the expression; other expressions can refer to this label in their *refer_to* slot, in order to reuse the expression
- *refer_to*: a symbol pointing to a label; the current expression can reuse another expression using the label in the *label* slot of the latter
- *type*: either *spec*, *cond* or *opt*; expressions that are continuation specifications have type *spec*, conditions have type *cond*, and only some internal expressions have type *opt*, as we will see in subsection 6.2.4
- *operator*: either a logical operator (*or*, *and* or *not*) that combines the subexpressions in slot *subexps*, or *atomic*, meaning that there are no subexpressions
- *subexps*: either a list with the names of the subexpressions that are combined by the logical operator (and their truth values), or (if the operator is *atomic*) the complete expression

Optional attributes

- *period*: in case the expression is a condition with a time annotation, this slot contains the numbers and units of the corresponding starting and finishing shifts

- *since*: in case the expression is a condition with a time annotation, this slot contains the name of the corresponding reference
- *explanation*: in case the expression contains an additional explanation, it can be stored within this slot

Internal attributes

- *value*: the current value of the expression, being either *true*, *false*, or *unknown*, depending of the evaluation result of the expression; default is *unknown*

Some example facts representing real expressions from the Jaundice protocol are the following:

```
(exp (name complete_Diagnostics-and-Treatment-hyperbilirubinemia) (operator or) (type cond)
(subexps complete_Diagnostics-and-Treatment-hyperbilirubinemia_1 unknown complete_Diagnostics-
and-Treatment-hyperbilirubinemia_2 unknown))
```

```
(exp (name complete_Diagnostics-and-Treatment-hyperbilirubinemia_1) (operator atomic) (type cond)
(subexps jaundice-clinically-significant equal no) (explanation "Follow this infant into routine clinical
supervision."))
```

5.3 Representation of domain definitions

We have already mentioned that variables, parameters and constants are part of Asbru Light. Hence, a way must be found to represent these entities. Since our interpreter will treat variables and parameters in exactly the same way, we have decided to design only one template for both of them. This template is shown in figure 18. It can be used to model all possible types of variables and parameters, no matter if they are constant or changeable, or global or local. It can also support the different types of abstraction and time annotations, as will be shown later on. With some modification, it could even be appropriate for representing arrays or lists. Only constants are represented in a different way. They are not parsed at all, but are defined within the code of the program.

(deftemplate value

```
  (slot left
    (type SYMBOL))
  (multislot right)
  (slot since
    (type NUMBER))
  (multislot old)
  (slot ref
    (type SYMBOL))
)
```

Figure 18. Final template for the representation of a variable or parameter in Clips.

The possible attributes a variable (or parameter) may have can only be divided into two classes, namely mandatory and optional attributes. However, these classes here have a slightly different meaning than they had earlier on because variables, as opposed to plans and expressions, do not serve as input for the Clips program. They do not exist as facts until the moment when they receive a value, mostly by means of an *ask* or *variable-assignment*, which is always during the execution of the Clips program. The consequence is that the class of mandatory attributes now represents the attributes a variable must have during the execution. The meanings of all attributes are discussed below.

Mandatory attributes

- *left*: a symbol defining the name of the variable
- *right*: one or (for a list) more symbols defining the value(s) of a variable
- *since*: a number containing the point of time when the variable was last modified
- *old*: one or more symbols defining the value(s) that the variable had before its last modification

Optional attributes

- *ref*: in case a variable refers to another variable (e.g. TSB-change that monitors the change in TSB-value), this slot contains the name of the main variable on which it is based

Note that the initial values of all variables are unknown. This means that all conditions based on certain values are initially unknown as well. Only when a variable has received a value (by means of an ask, an assignment, or whatsoever), it can be used by the program. Some examples of facts representing variables are:

```
(value (left drugs) (right drug1 drug2 drug3) (since 10.554) (old nil))
(value (left doses) (right 210 410 610) (since 27.899) (old 200 400 600))
(value (left possibility-of-cholestatic-disease) (right no) (since 30.352) (old nil))
(value (left TSB-value) (right 5) (since 32.175) (old 3))
(value (left TSB-change) (right -2) (since 32.175) (ref TSB-value))
(value (left age-child) (right 30) (since 33.349) (old nil))
```

However, this representation is not satisfactory for representing the whole notion of abstraction. Within Asbru, the abstraction from qualitative data to quantitative data is represented by a mapping to a so-called scale. Speaking in Clips terms, we decided to translate this information into facts of the following kind:

```
(reference age age-child)
(scale age-scale day1 day2 day3 later)
(limits age age-scale 0 24 48 72 1000)
```

The meaning of the first fact is that the value of the qualitative parameter 'age' will be abstracted from the quantitative parameter 'age-child'. The other two facts could best be read together. They mean that the value 'day1' concerns the patients between 0 and 24 hours old, 'day2' concerns those between 24 and 48, and so on.

In some special cases, an abstraction is based on more than one parameter. For these cases, we introduce facts of the form *special-limit*. Consider the following example, stating that (bilirubin = observation) when the TSB-value is between 0 and 12 AND the extra condition (age = day2) holds:

```
(reference bilirubin TSB-value)
(scale bilirubin-scale observation phototherapy-recommended phototherapy-normal phototherapy-intensive transfusion)
(special_limits bilirubin bilirubin-scale age day2 0 12 15 20 25 1000)
```

Note that we have not designed a special template for the representation of the facts concerning abstraction. This is because there is no need to separate the attributes by means of slots.

5.4 Representation of time

While explaining the templates for expressions and values, we have already demonstrated some design decisions related to time annotations in Asbru. However, another important design decision within this project is how to implement the general notion of time. In other words, how will a clock tick be simulated? Section 4.1 was already partly dedicated to this subject. As explained there, it will not be necessary to implement a program that simulates an exact guideline execution second for second. On the other hand, we have already decided that we want to support cyclical plans and temporal conditions (subsection 4.2.3), so at least any notion of time is inevitable. The solution for this problem became the following. The idea is to make use of Clips' built-in function *time()*, which returns the amount of seconds that have passed since the start of the program. Using the construction, the interpreter could for instance simulate a cyclical plan by activating it every *n* seconds. One problem is though that many cyclical plans are repeated e.g. every 2 weeks (= 1209600 seconds) and no Asbru plan designer would have the patience to wait that long. This is why we introduce some new facts called *multiplication factors*, which have the following structure:

```
(weeks_multiply_factor 5)
```

This fact stands for the translation from weeks (in guideline time) to seconds (in simulation time). Hence, one week is simulated as 5 seconds. Multiplication factors are stored within the code of the Clips program, so that they can be modified by the user at any time. Like for the abstraction facts, there was no need to define a special template for them.

6. The interpreter

The subject of this chapter is the interpreter itself. Since the program consists of two different parts, a Prolog and a Clips program, the contents of these programs will separately be treated, in section 6.1 and 6.2. Section 6.3 illustrates the behaviour of both programs by means of some example output.

6.1 The Prolog parser

The first part of the interpreter, the Prolog parser, is the part that takes an Asbru protocol written in XML as input, and translates this into Clips facts like the ones discussed in the previous section. This program, like all programs written in Prolog, consists of different predicates called *clauses*[27]. A short introduction to Prolog is given in Appendix A. In subsection 6.1.1, a high-level description of our Prolog program, as well as a description of the top level clause will be given. As an illustration, subsection 6.1.2 will examine the clauses that parse plans.

6.1.1 Top level structure

The top level predicate of the program is a clause named *start* (see figure 19). It is by executing this clause, together with two arguments specifying the names of the input and output files, that the program is started. Its most important part is the predicate *load_xml_file(InFile, Tree)*, which is a built-in predicate of Prolog version 4.0.0[25]. This predicate receives as input a complete XML file. Subsequently, it parses this file and translates its elements to a tree in Prolog syntax. In this translation, the nested structure of the XML file stays intact. Figure 20 shows an example of how the predicate would translate an arbitrary XML file.

```
Start(InFile, OutFile):-
    tell(OutFile),
    load_xml_file(InFile, [element('plan-library', [], [element('library-info', _, _), element('domain-defs', [],
        [element('domain', _, Defs])), element('plans', [], [element('plan-group', [], Plans])])])),
    write('(deffacts facts'), nl,
    init,
    parse_all_defs(Defs),
    parse_all_proper_plans(Plans),
    parse_all_fictive_plans,
    parse_all_expressions,
    parse_all_on_aborts,
    write(')'), nl,
    told.
```

Figure 19. Prolog predicate 'start'. Top level clause of the Prolog parser.

XML file	Prolog tree
<pre><el-1> <el-2/> <el-3/> </el-1></pre>	<pre>[element(el-1, [], [element(el-2, [], []), element(el-3, [], [])])]</pre>

Figure 20. Translation from XML file to Prolog tree.

Hence, the behaviour of the whole program is as follows. Firstly, the specified output file is opened by means of the predicate *tell*. After that, *load_xml_file* reads in the specified input file and translates it into a tree as described in figure 20. After the translation has been made, our Prolog program still has to parse different branches of the resulting tree, and turn them into correct Clips facts. This is done in five main predicates, in which the Asbru concepts domain definitions, plans, fictive plans, expressions and on_aborts are treated. By this treating we mean that the predicates prune the necessary information from the tree, and write it to the output file in Clips syntax. More detailed examples are given in the next subsections. Finally, the output file is closed by means of the *told* predicate.

6.1.2 Parsing of plans

As can be deduced from figure 19, the predicate *parse_all_proper_plans* receives as argument the complete branch of the Prolog tree that is a child of the element ‘plan-group’. This branch corresponds to a list containing all plans of the Asbru protocol. *Parse_all_proper_plans* is also a recursive predicate that treats all these plans one by one, until the list is empty. The way the plans are treated is rather straightforward. The program simply parses all the relevant parts (name, conditions, continuation specification, plan_body), thereby immediately writing the corresponding part of the Clips fact to the output. For instance, when a particular predicate parses the body of a plan and discovers that this plan has the attribute *wait-for-optional-subplans* set to *yes*, the text ‘(wait_optional yes)’ is written to the output. Besides, apart from the facts representing proper plans, the predicate *parse_all_proper_plans* also produces another fact. This is a fact of the form (start x), where x is a plan name. It is done in order to tell the Clips program which plan is the top level plan, and should thus be executed first.

As an illustration of the way the program checks for relevant parts of plans, figure 21 shows (part of) the predicate *parse_body*, which parses the body of a plan, provided to the predicate in the ‘tree’ syntax. This clause checks if the body has *parallel subplans*, with the attributes *wait-for-optional subplans* and *retry-aborted subplans* set to *no*. If so, this information is immediately written to the output in Clips syntax. The clause continues by executing the predicate *parse_spec*, which parses the continuation specification. Note that *parse_body* consists of more clauses than the one shown. These other clauses check for plans with other types of bodies.

```

parse_body(element('subplans', [ 'type'='parallel', 'wait-for-optional-subplans'='no', 'retry-aborted-
subplans'='no'], Rest), Name):-
    write(' (type ',
    write('do_parallel'),
    write(')'),
    parse_spec(Rest, Name).

```

Figure 21. Part of Prolog predicate ‘*parse_body*’, translating a parallel plan-body into Clips syntax.

Notice that the only output this clause would produce is the following: “(type do_parallel)”, which is just a small part of a complete Clips fact. In other words, a single Clips fact will be generated by several Prolog clauses together. However, one important remark should be added, and this concerns fictive plans. That is, when a complete fictive plan would be written to the output right after being parsed, it would end up somewhere in the middle of its parent plan, resulting in something like this:

```
(plan (name A) (type do_sequential) (subplans ((plan (name B) (type user_performed))) )
```

However, we had just decided to represent fictive plans as separate Clips facts. This is why we chose to have the program temporarily assert all parts of the tree that represent fictive plans as Prolog facts. This way, they can be parsed later (by the predicate *parse_all_fictive_plans*). The same procedure is repeated for conditions and on-aborts, because these will also be represented as separate facts. What the predicate *parse_all_fictive_plans* does is to search for the stored Prolog facts representing fictive plans, parse them and finally retract them. In order to parse them, more or less the same predicates are used as in case of the proper plans. The only difference is that the predicate for parsing the conditions can be skipped, because fictive plans never have conditions.

6.2 The Clips program

The Clips program is the second part of the interpreter. Its input is a file of facts in the representation introduced in chapter 5, produced by the Prolog program. Then, it uses this to simulate the execution of the corresponding Asbru protocol, thereby producing a trace of actions as output. Like all programs in Clips, it consists of a set of *deftemplates*, *deffacts*, *deffunctions* and *defrules*[27]. An introduction to Clips is given in Appendix B.

The general behaviour of the Clips program is as follows. First, the top level plan is activated automatically, because the input file contains one fact of the form (start x). After that, plans may activate each other by producing new facts of the form (start x), where x is one of their subplans. As soon as a plan is activated, there is a constant monitoring loop that checks if they should switch to a new state. This checking is based in the evaluation of the plan's conditions. Meanwhile, all important steps are written to the output. By important steps we mean all state transitions and all actions done by atomic plans, such as a variable assignment. The program finishes as soon as no more state transitions or plan activations are possible. Note that many defrules have different *salience* priorities, which implies that they are fired in a predefined order. For instance, the evaluation of expressions should have a higher priority than the checks for conditions, because a condition can only be checked if it has a truth value.

The following subsections will explain some of the program's most important defrules, as well as some remarkable choices that were made during implementation. They will respectively discuss state transitions, evaluation of expressions, execution of plans, abstraction, pre-processing, and the interpreter output.

6.2.1 State transitions

Given the representation of plans, defined in section 5.1, the implementation of the different state transitions of Asbru Light (which correspond to a restriction of figure 3, namely the transitions between the states *considered*, *ready*, *rejected*, *activated*, *aborted* and *completed*) is a rather straightforward process. The fact that the state of each plan can be verified all the time, by having Clips check the different *state* slots, allows us to write different Clips rules for all existing transitions in the diagram. The useful aspect of this solution is that Clips constantly keeps checking the applicability of all the rules. This means that as soon as a certain condition of a plan becomes true, the program automatically fires the corresponding rule, which was exactly the reason for choosing Clips. We distinguish two types of state transitions, namely normal transitions and those produced by propagation.

Normal transitions

We have already pointed out that each single transition of our Asbru Light selection of the state transition diagram will be implemented by one or more special defrule(s) in Clips. The complete set of defrules representing state transitions is included in Appendix B. An example of such a defrule, which represents the normal transition from state *ignored* to *considered*, is shown in figure 22.

```
(defrule consider_plan_1
  (declare (salience 5))
  ?f <- (plan (name ?name) (type ?type) (proper yes) (state ignored) (invoked_by $?invokers)
          (your_turn yes))
  (start ?name)
  =>
  (printout t ?name)
  (printout t " considered" crlf)
  (modify ?f (state considered))
  (assert (schedule $?invokers))
)
```

Figure 22. Clips defrule 'consider_plan_1', representing the transition of proper plans from state ignored to considered.

This defrule is probably one of the first rules that will fire when running the program, because its if-part already holds from the start. This is because there is always a toplevel plan, i.e. a proper plan with name *?name* and state *ignored*, for which the fact (start ?name) holds. As can be seen in the figure, the then-part of the rule expresses that the plan's state transition is printed, and the state of the fact is adapted accordingly. The rest of the

statements are not relevant for the current explanation, so they will not be treated here. One final remark is that, for those plans that are fictive (and thus have no conditions), there is a rule similar to *consider_plan_1*, but now for the immediate transition to state *activated*. This rule is called *consider_plan_2*. This transition does of course not exist in the original state transition diagram, because this is based on proper plans only.

Like for the transition rule from state *ignored* to *considered*, comparable defrules have been written for the other transitions of figure 3. For instance, there are several defrules for the transition from state *considered* to *ready* (remember that *possible* does not exist in Asbru Light). These rules check if the plan has a true (or empty) filter precondition. If so, the plan switches indeed to state *ready*. If not, the plan normally switches to state *rejected* at once, according to the definition of the filter precondition. However, there are situations when this transition should not take place yet. For instance, when the plan in question is the subplan of a parallel plan, and this parallel plan has another child in state *considered*, whose filter condition is true. In these cases, the rule should not fire until the other child is in state *ready*. In order to solve this problem, we gave the transition rule a lower priority by means of a *salience* statement.

The transition from state *ready* to *activated* was relatively easy to model. For this, the activate mode has to be checked. If it is automatic, the transition can take place without any problems. If it is manual, the user is first asked if the plan is allowed to start. If this is the case, the transition can still take place. If not, the plan switches to state *rejected*.

Another very simple transition is the one from state *activated* to *aborted*, based on the abort condition. If it is true, the transition can take place immediately. If it is false, the rule is not fired so nothing happens.

The last transition modelled in this part of the program is the transition from state *aborted* to *ignored*. This rule should fire when a plan is in state *aborted*, and its (active) parent has the attribute *retry-aborted-subplans* set to yes. In this case, not only the plan in question should be reactivated, but also all of its children. Indeed, this transition is not part of the original state transition diagram, because it is not based on a condition. Nevertheless, it should be modelled anyway, because it is the only way to reactivate aborted subplans.

Propagation

As mentioned earlier, propagation occurs whenever a parent plan must switch to another state because of a transition of its child, or vice versa. Within Asbru, there are three kinds of propagation, which are all supported by our program. First, one kind of propagation concerns the completion of parent plans. This is the case whenever all of the following conditions hold for a certain plan:

- 1) its continuation specification is satisfied
- 2) its complete condition is true or absent
- 3) its wait-for-optional-subplans attribute is either *no*, or it is satisfied (which means that all subplans have finished executing)

Hence, our propagation defrules constantly check for the above situation. And if it is the case, the parent plan switches to state *completed*. One of these rules is shown in figure 23.

(defrule complete_propagation_1a

```
(declare (salience 70))
?f <- (plan (name ?name) (cont_spec ?spec) (wait_optional no) (complete_cond ?cc) (state activated))
(exp (name ?cc) (value true))
(exp (name ?spec) (value true))
=>
(printout t " ")
(printout t ?name)
(printout t " completed" crlf)
(modify ?f (state completed))
)
```

Figure 23. Clips defrule ‘complete_propagation_1a’. This rule fires whenever the continuation specification is satisfied, the complete condition is true, and wait-for-optional-subplans is no.

The second kind of propagation concerns the following: whenever it turns out that a parent plan's continuation specification is unsatisfiable (which means that one of its mandatory subplans has aborted or rejected, and will never be reactivated), this plan switches to state *aborted*.

The third kind of propagation concerns the abortion of subplans. It is the case whenever a parent plan aborts or completes. In that case, all of its active subplans should switch to state *aborted*, and so we implemented a defrule that reflects this course of things. In addition, all subplans that had not started yet should be prevented from starting at all. That is why they will switch to state *aborted* as well.

6.2.2 Evaluation of expressions

The above subsection has described how plans in the Clips program may switch from any state to another. In most of the times, these transitions are based on the truth value of a certain condition (or expression). The representation of expressions has already been explained in section 5.2. However, a next question has not been answered yet: how does the program evaluate the truth value of expressions? In order to find an answer to this question, we will respectively treat atomic expressions, composed expressions, and continuation specification.

Atomic expressions

Since expressions can be composed as well as atomic, we have to create rules for the evaluation of both of them. We will first explain those rules that evaluate an atomic expression. For all conditions of Asbru Light (subsection 4.2.2), we have implemented defrules that evaluate them, and adapt their value accordingly. Since the truth values of expressions may change at any time during the execution, the program must keep on checking them continuously. This procedure is illustrated best by showing some real Clips code. The rule that determines if expressions of type *equal* are true, is shown in figure 24. This rule checks if there exists a variable or parameter called ?left, whose value is ?right. If so, the truth value of the atomic expression is changed to *true*. Furthermore, the priority of this rule is high, so that it always fires as soon as possible. Similar rules have been implemented for the truth values *false* and *unknown*, so that the slot *value* of each expression is kept up-to-date. Apart from that, note that the facts representing the variables are updated at the very same moment that these variables receive their values, i.e. within an *ask* or *assignment* rule (see next subsection).

```
(defrule eval_true_condition_1
  (declare (salience 75))
  ?f <- (exp (type cond) (operator atomic) (subexps ?left equal ?right) (since nil) (value ?value))
  (test (neq ?value true))
  (value (left ?left) (right ?right))
  =>
  (modify ?f (value true))
)
```

Figure 24. Clips defrule 'eval_true_condition_1'. Determines if atomic expressions of type 'equal' are true.

Note that a comparable set of rules has been written for those atomic expressions that contain a time annotation. These rules have exactly the same structure, but in addition, there is a construction within the rule's if-part that checks if the time annotation is satisfied. An example of such a check is the verification if the value of the *since* slot of an expression is greater than the current time, or than the *since* slot of a certain plan (see the example in figure 25).

```

(defrule eval_true_condition_time_1a

  (declare (salience 75))
  ?f <- (check_time)
  ?g <- (exp (type cond) (operator atomic) (subexps ?left equal ?right) (period ?period) (since start)
        (value ?value))
  (test (neq ?value true))
  (start_time ?st)
  (value (left ?left) (right ?right))
  (test (> (- (time) ?st) ?period))
  =>
  (modify ?g (value true))
  (retract ?f)
)

```

Figure 25. Clips defrule ‘eval_true_condition_time_1a’. Determines if atomic expressions of type ‘equal’, with a time annotation, are true.

Composed expressions

Given the fact that our program is able to determine the truth value of atomic expressions, we must now find a way to evaluate composed expressions. Consider figure 26, showing a rule that fires when a certain expression has a subexpression with name *?name* and truth value *true*. In that case, the truth value of that plan is also set to *true* within the slot *subexps* of its parent expression. Similar rules have been implemented for the truth values *false* and *unknown*, so that the slot *subexps* of each expression is kept up-to-date.

```

(defrule subexpression_true

  (declare (salience 75))
  ?f <- (exp (subexps $?dummy1 ?name ?value $?dummy2))
  (exp (name ?name) (value true))
  (test (neq ?value true))
  =>
  (modify ?f (subexps $?dummy1 ?name true $?dummy2))
)

```

Figure 26. Clips defrule ‘subexpression_true’. Switches the value of a subexpression to true.

In addition to the rules that update the *subexps* slots, we created defrules that update the truth value of the whole composed expression. Consider figure 27, providing a rule that fires when an expression with operator *or* contains a true subexpression. In that case, the truth value of the parent expression is set to *true* as well. Similar rules have been implemented for the truth values *false* and *unknown*, and for the operators *and* and *not*, so that all possible kinds of combined expressions are covered.

```

(defrule or_expression_true

  (declare (salience 75))
  ?f <- (exp (operator or) (subexps $?dummy1 true $?dummy2) (value ?value))
  (test (neq ?value true))
  =>
  (modify ?f (value true))
)

```

Figure 27. Clips defrule ‘or_expression_true’. Determines if a composed expression with operator ‘or’ is true.

Continuation specifications

The template *exp* can be used to model conditions as well as continuation specifications. In case of conditions, the parameters combined by the logical operators are atomic expressions. As for the continuation specifications,

the parameters that are combined are plan names. Hence, for these plan names there should be a mechanism that assigns the truth value *true* to a completed plan. The corresponding defrule is shown in figure 28. Likewise, there are rules that set an aborted or rejected plan to *false* and an ignored plan to *unknown*.

```
(defrule subplan_true

  (declare (salience 75))
  ?f <- (exp (type spec) (subexps $?dummy1 ?name ?value $?dummy2))
  (plan (name ?name) (state completed))
  (test (neq ?value true))
  =>
  (modify ?f (subexps $?dummy1 ?name true $?dummy2))
)
```

Figure 28. Clips defrule ‘subplan_true’. Switches the value of a completed subplan to true.

6.2.3 Execution of plans

The most complicated part of Asbru is the plan body, because it can be used to express a very large amount of control structures. As a consequence, the plan body is also the most difficult part of the language to implement. According to the template shown in figure 15, a plan may have one of the following types: *do_sequentially*, *do_parallel*, *do_any_order*, *do_unordered*, *cyclical*, *if_then_else*, *ask*, *assignment*, *manipulation*, *display*, *user_performed*, or *do_for_each*. This subsection will give a short explanation of how all of these types are implemented. It will in particular discuss the first four of them, because they are the most complicated. Note that most of our implementations are conforming to the formal semantics for Asbru. However, in some cases that we particularly mention, there were reasons to deviate from the semantics.

Do_sequentially

Of the four main structures for controlling the dynamics of subplans, the sequential plans are the easiest to implement. The idea is that, as soon as a plan of type *do_sequentially* is activated, the first plan of its *subplans* slot is allowed to start as well. This is done by means of the assertion of a (start ?name) fact, as can be seen in the first defrule of figure 29. As soon as this plan has completed, rejected or aborted, the second subplan is allowed to start, and so on. This procedure can be seen in the second defrule of figure 29, where the (*ignored*) plan with name ?seq2 is the next plan to be started.

```
(defrule do_sequentially_first

  (plan (type do_sequentially) (subplans ?seq1 $?) (state activated) (invoked_by $?invokers)
    (your_turn yes))
  ?f <- (plan (name ?seq1) (state ignored))
  =>
  (assert (start ?seq1))
  (modify ?f (invoked_by $?invokers))
)

(defrule do_sequentially_rest

  (plan (type do_sequentially) (subplans $? ?seq1 ?seq2 $?) (state activated) (invoked_by $?invokers)
    (your_turn yes))
  (or (plan (name ?seq1) (state completed)) (plan (name ?seq1) (state rejected)) (and (plan (name ?seq1)
    (state aborted)) (not (on_abort ?seq1 ?))))
  ?f <- (plan (name ?seq2) (state ignored))
  =>
  (assert (start ?seq2))
  (modify ?f (invoked_by $?invokers))
)
```

Figure 29. Clips defrules ‘do_sequentially_first’ and ‘do_sequentially_rest’.

Do_parallel

As opposed to sequential execution, parallel execution of plans is a very difficult to implement. Since true parallelism (which means that several subplans really run at the very same point of time) is impossible to program without a multiprocessor, we decided to simulate it in a synchronous way. By this simulation we mean that each subplan does one step at a time. For instance, when a parallel plan with two subplans would be activated, the children's behaviour could be the following: (subplan-1 considered) – (subplan-1 considered) – (subplan-1 ready) – (subplan-2 ready) – (subplan-1 rejected) – (subplan-2 activated), and so on. The defrules that implement the parallel behaviour are shown in figure 30. At first sight, they might seem very similar to the rules for sequential plans, but there is one big difference. In order to explain this difference, we first need to introduce the notion of *invoker*.

(defrule do_parallel_toplevel

```
(plan (type do_parallel) (subplans ?par1 $?) (state activated) (invoked_by none))
?f <- (plan (name ?par1) (state ignored))
=>
(assert (start ?par1))
(modify ?f (invoked_by ?par1 unknown))
)
```

(defrule do_parallel_subplan

```
(plan (type do_parallel) (subplans ?par1 $?) (state activated) (invoked_by $?invokers) (your_turn yes))
(test (neq (nth$ 1 $?invokers) none))
?f <- (plan (name ?par1) (state ignored))
=>
(assert (start ?par1))
(modify ?f (invoked_by $?invokers ?par1 unknown))
)
```

Figure 30. Clips defrules 'do_parallel_toplevel' and 'do_parallel_subplan'.

Consider figure 31. When a parallel plan has two subplans, in this example plan B and plan C, then we say that all nodes of the subtree of plan B (including plan B itself), are *invoked* by plan B, as soon as they are activated. Hence, in the end plan B is the *invoker* of the plans B, B1 and B2. Likewise, the whole branch of the tree below plan C is invoked by plan C. Furthermore, if a plan has more than one parallel ancestor, it has also several invokers. Hence, the invokers of B1 are both B and B1. All invokers of all plans are continuously stored and updated in the slot *invoked_by*. This can be seen in the last statements of both rules in figure 30. The idea is that, by means of these invokers, the program can determine if the plan in question is allowed to do a step. In other words, if it is "its turn". This is only the case if the truth values of all invokers of the plan are set to *true*, and in that case, its slot *your_turn* is set to *yes*. An invoker is only set to *true* when it is the first element of its parent's *subplans* slot, because only then, the corresponding branch of the tree is allowed to do a step. (This is because the parallel plans always only activate their first subplan, see figure 30). Note that we have implemented a whole set of rules that constantly update all *your_turn* slots. These rules are not shown here, but they are very much comparable to those that evaluate expressions in the last subsection. What they do in broad outline is to set *your_turn* to *yes* when all invokers are true, or if there are no invokers. Otherwise, *your_turn* is set to *no*. And, with the notion of "your turn", the implementation of the parallel behaviour is easier, because, as soon as a certain subplan has done a step, all its invokers are being rescheduled. By *scheduling* is meant that the program places the plans at the end of the *subplans* slot they are part of, which makes that it is not their turn anymore. In fact, the *subplans* slot is thus used as a queue here. Note that none of the above values is ever reset. However, all of them are continuously updated.

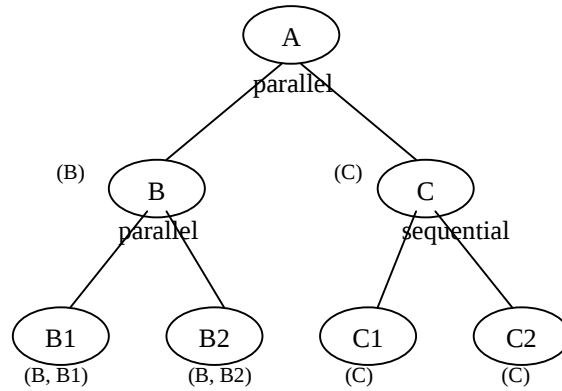


Figure 31. Hierarchy of plans. For each plan, its invokers (in slot “invoked_by”) are mentioned between brackets.

This complicated behaviour can be illustrated better by means of an example trace. Consider the following example, for which the plans of figure 31 are used. Hence, A and B are parallel plans. C is a sequential plan, and the rest are atomic (e.g. user-performed) plans. In the beginning, we have the following situation:

```

(plan (name A) (type parallel) (subplans B C) (state activated) (invoked_by none) (your_turn yes))
(plan (name B) (type parallel) (subplans B1 B2) (state ignored) (invoked_by none) (your_turn yes))
(plan (name C) (type sequential) (subplans C1 C2) (state ignored) (invoked_by none) (your_turn yes))
(plan (name B1) (type user-performed) (state ignored) (invoked_by none) (your_turn yes))
(plan (name B2) (type user-performed) (state ignored) (invoked_by none) (your_turn yes))
(plan (name C1) (type user-performed) (state ignored) (invoked_by none) (your_turn yes))
(plan (name C2) (type user-performed) (state ignored) (invoked_by none) (your_turn yes))

```

Thus, plan B is allowed to perform a step, because it takes up the first position in the list of subplans of plan A. Hence, plan B is activated and immediately rescheduled, which makes that it is not its turn anymore. In addition, the name of plan B is added to its own list of invokers, in accordance with defrule *do_parallel_toplevel*:

```

(plan (name A) (type parallel) (subplans C B) (state activated) (invoked_by none) (your_turn yes))
(plan (name B) (type parallel) (subplans B1 B2) (state activated) (invoked_by B false) (your_turn no))
(plan (name C) (type sequential) (subplans C1 C2) (state ignored) (invoked_by none) (your_turn yes))
(plan (name B1) (type user-performed) (state ignored) (invoked_by none) (your_turn yes))
(plan (name B2) (type user-performed) (state ignored) (invoked_by none) (your_turn yes))
(plan (name C1) (type user-performed) (state ignored) (invoked_by none) (your_turn yes))
(plan (name C2) (type user-performed) (state ignored) (invoked_by none) (your_turn yes))

```

After that, plan C does the same thing. Hence, it is re-scheduled behind plan B, which makes that it is the turn of plan B again:

```

(plan (name A) (type parallel) (subplans B C) (state activated) (invoked_by none) (your_turn yes))
(plan (name B) (type parallel) (subplans B1 B2) (state activated) (invoked_by B true) (your_turn yes))
(plan (name C) (type sequential) (subplans C1 C2) (state activated) (invoked_by C false) (your_turn no))
(plan (name B1) (type user-performed) (state ignored) (invoked_by none) (your_turn yes))
(plan (name B2) (type user-performed) (state ignored) (invoked_by none) (your_turn yes))
(plan (name C1) (type user-performed) (state ignored) (invoked_by none) (your_turn yes))
(plan (name C2) (type user-performed) (state ignored) (invoked_by none) (your_turn yes))

```

The next step plan B performs is the activation of its first subplan, plan B1. Both B and B1 are scheduled, which makes that it is the turn of plan C:

```

(plan (name A) (type parallel) (subplans C B) (state activated) (invoked_by none) (your_turn yes))
(plan (name B) (type parallel) (subplans B2 B1) (state activated) (invoked_by B false) (your_turn no))
(plan (name C) (type sequential) (subplans C1 C2) (state activated) (invoked_by C true) (your_turn yes))
(plan (name B1) (type user-performed) (state activated) (invoked_by B false B1 false) (your_turn no))

```

```
(plan (name B2) (type user-performed) (state ignored) (invoked_by none) (your_turn yes))
(plan (name C1) (type user-performed) (state ignored) (invoked_by none) (your_turn yes))
(plan (name C2) (type user-performed) (state ignored) (invoked_by none) (your_turn yes))
```

The rest of the trace is obvious. Plan B and C will be allowed to do one step at a time, until one of them finishes. Note that plan C is a sequential, and not a parallel plan. This means that its children will ‘inherit’ its exact list of invokers. However, without the addition of plan C. This ‘inheritance’ can be found in the last statements of figure 29 (showing sequential rules).

Do any order

One would expect that the implementation of subplans being executed in any order, in accordance to the explanation of subsection 3.1.1, would lead to some problems, because the resulting behaviour is non-deterministic. The non-determinism is caused by the fact that the order of the subplans is not specified. That is why we decided to implement the any-order subplans as follows. As soon as the plan is activated, the user is asked to type in its desired order, which will thereupon be executed as if the plan were sequential. This behaviour can be seen in figure 32. As soon as the plan is activated (and it is allowed to do a step!), the deffunction *ask_order* is called, with as arguments the list of subplans. This function asks the user to type in the order he or she desires. This should be done by simply typing a number for each subplan, separated by spaces. For instance, a list of four subplans that should be executed in inverse order, would be typed in as “4 3 2 1”. Afterwards, *ask_order* immediately changes the fact into a fact of type *do_sequentially*, thereby changing the order of the subplans, by means of a *modify* statement. We are aware of the fact that our solution deviates from the formal semantics of Asbru. However, we have chosen for this solution, for the reason mentioned above. Another, even more important motivation is that the intended user group is knowledge engineers debugging Asbru plans. These users are often interested in what happens with particular orderings of any_order plans. Our implementation offers them the opportunity to choose which ordering will be taken by the interpreter.

```
(defrule do_any_order
```

```
  ?f <- (plan (name ?name1) (type do_any_order) (subplans $?subplans) (state activated)
            (your_turn yes))
  =>
  (ask_order ?f $?subplans)
)
```

Figure 32. Clips defrule ‘do_any_order’.

Do unordered

In addition to the above three types of plans, the unordered plans are relatively easy to implement. This is because there is no need for the program to take account of many conditions before the rule may fire. The only conditions that must hold is that a certain plan with type *do_unordered* is in state activated, is allowed to do a step, and has a subplan in state ignored, see figure 33. Also note that the position of the subplan within the slot *subplans* is not important, since it does not matter at all at what moment it is activated. This makes indeed that the rule may fire at many different moments. Whenever there are two subplans that are both allowed to be activated, the Clips mechanism for selecting rules will determine in which order this will be done. Of course, if the rules have different priorities by means of *salience* statements, the selection is logical. But is this is not the case, the selection is based only on the order in which the relevant facts were asserted. By default, Clips picks the rule that uses the ‘youngest’ facts. However, this setting may be changed by the user. Note that this behaviour also allows the parallel execution of unordered plans. For example, consider an unordered plan with two subplans. Suppose that the filter condition of one of them is true, and that the filter condition of the other is unknown. Hence, the first subplan starts executing. Now suppose that the filter condition of the second subplan becomes true during the execution of the first. Then this second subplan is suddenly activated while the first one was halfway its execution.

```
(defrule do_unordered
```

```
  (plan (type do_unordered) (subplans $? ?unor1 $?) (state activated) (invoked_by $?invokers)
    (your_turn yes))
  ?f <- (plan (name ?unor1) (state ignored))
  =>
  (assert (start ?unor1))
  (modify ?f (invoked_by $?invokers))
)
```

Figure 33. Clips defrule ‘do_unordered’.

Cyclical

The most important aspect of the cyclical plan is the fact that it involves the notion of time. Therefore, we decided to use Clips’ built-in function *time()* for the implementation. Imagine for instance a plan that should do something every *n* seconds. Then, the only thing that the defrule in question has to do is to check if the current time is a multiple of *n*, before the subplans may become activated. Note that if the cyclical plan has several subplans, we treat all these subplans as one block. This means that, even if the cyclical plan runs in parallel with another plan, all subplans are still activated after each other. This is another point where our implementation differs from the original Asbru semantics. We made this choice because the idea of the cyclical plan was to perform actions at predefined time points. Thus, it would not make sense to wait for your turn after each step, because then the time point would already have passed. In addition, it would not make much difference, because most cyclical plans in the Jaundice and Diabetes protocol are loops of simple actions like asks. For the rest, caution should be taken when such an action is activated for the second time, because normally, the plan is already in state completed then. Hence, the whole plan and all of its subplans should be “reset” to state ignored. The defrule that implements the cyclical plan is shown in figure 34. Since it contains a lot of uninteresting and complex code, we will not discuss it in detail.

```
(defrule cyclical
```

```
  (plan (type cyclical) (repeat ?m ?n) (subplans ?name) (state activated) (invoked_by $?invokers) (since
    ?since))
  ?f <- (plan (name ?name) (state ?state) (subplans $?subplans))
  (test (or (eq ?state ignored) (eq ?state completed) (eq ?state aborted)))
  (test (eq 0 (mod (integer (* 1 (- time) ?since))) (integer (/ (+ ?m ?n) 2))))
  ; Is (rounded) difference of plan start time and current time a product of the average of m and n?
  ?g <- (last_cycle ?lc)
  (test (neq (integer (* 1 (- time) ?since))) ?lc))
  ; Action has not been performed this time yet?
  =>
  (assert (last_cycle (integer (* 1 (- time) ?since))))
  (retract ?g)
  (printout t ?name)
  (printout t " (cycle)" crlf)
  (modify ?f (state activated) (since (time)))
  (assert (reset $?subplans))
)
```

Figure 34. Clips defrule ‘cyclical’.

If_then_else

The if_then_else plans are rather unique plans. They cannot really be considered as plans with subplans, but they are not really atomic either. In fact, the only thing they do (when they are allowed to do a step) is to verify if a certain condition holds. And based on the result, one of the two subplans is activated.

Atomic plans

The plans that are called atomic within this document have one of the following types: *ask*, *assignment*, *display*, or *user_performed*. What they all have in common is that they do not activate any other plans. Hence, as soon as they are allowed to do a step, they actually do this and then they complete immediately. The way their simple actions are implemented is mentioned below for each plan:

- *ask*: ask the user to enter a parameter by means of a special deffunction, and assert a fact containing the resulting value immediately
- *assignment*: assign a fact containing the parameters in question. In addition, the assignments are written to the output, so that the user knows the new values of the parameters.
- *display*: display the string the plan has in its *subplans* slot, by means of a *printout* statement
- *user_performed*: print the name of the plan, ask the user if it has completed or aborted by means of a special deffunction, and adapt its state accordingly

6.2.4 Abstraction

The Clips program incorporates special rules for *abstraction*. As explained earlier, abstraction involves the conversion from quantitative data to qualitative data. Hence, the abstraction defrules check if there is a parameter for which a *scale* exists. If so, the corresponding value is abstracted, and the result is stored in a new fact as being the value of the reference parameter. For instance, when Clips finds out that the quantitative parameter ‘age-child’ is 30, the value of the reference parameter ‘age’ is stored as ‘day2’.

A concept slightly different from the abstraction, is the notion of *change*. There are some parameters, such as “TSB-change”, that are based on the last two values of another parameter. “TSB-change”, for instance, should be equal to the current value minus the last value of the parameter “TSB-value”. In our Clips program, we have implemented a defrule that calculates all change values whenever needed. For this, it is necessary that the last values of all parameters are stored and adapted continuously. The slot *old* within the *exp* template was reserved for this (figure 17), so that this can be done in a straightforward way when assigning new values.

6.2.5 Pre-processing

One very important part of the Clips program that has not been mentioned yet, is the set of *pre-processing* rules. These are defrules that should fire before (and not during!) the simulation of the guideline. This is why they all have a *salience* of 300. In fact, they perform operations that could have been done by the Prolog program as well, but are easier to do in Clips. The most lucid example is the pre-processing of the shortcuts for continuation specifications. These shortcuts, already mentioned in subsection 3.1.1, exist in three different types, namely *all*, *one* and *none*. This makes that the Prolog program is allowed to produce a plan with e.g. the following slot (cont_spec all). What the pre-processing rules do is to replace this ‘all’ by an appropriate continuation specification. In this case, this would be an AND-expression with all of the plan’s subplans as subexpressions. The same procedure could be followed for the other shortcuts.

Besides the shortcuts just mentioned, the program contains many other pre-processing rules. An example of is the pre-processing of time annotations. We have already introduced *multiplication factors* in section 5.4, which reflect the translation from guideline time to simulation time. Since all temporal values the input facts contain are initially “in guideline time” (e.g. 10 weeks), the translation to simulation time (e.g. 100 seconds) should be made for all of these facts. And this is the task of some pre-processing rules. What they do is to check for all time annotations in any type of plan or expression, and simply replace the value by its product with the multiplication factor.

One last type of pre-processing rules is the pre-processing of what we call *optional specifications*. These are concepts that do not exist within Asbru, but that we have invented ourselves for programming ease. We needed them for those plans whose *wait-for-optional-subplans* attribute is set to *yes*. For these plans, the normal continuation specification is not enough, because it is not the only condition that should hold before they can complete. In addition, all non-mandatory subplans should finish (i.e. complete, abort or reject) as well before they can complete. And this is why we invented the optional specifications. They have the same structure as the continuation specification, but they are only satisfied when all subplans (also the non-mandatory ones) have completed, aborted or rejected. Hence, the creation of such an optional specification is the task of one of our pre-

processing rules. The evaluation of this specification is of course done by some special rules, similar to those that evaluate the normal specification.

6.2.6 Output

The last important aspect with respect to the Clips program concerns the format that its output has. The easiest thing to implement is of course to simply print all state transitions, variable assignments and other important steps in free text. However, this is not the most well-organized format, from the user's point of view. That is why we examined the possibility of translating the output to a new XML-tree, thus the same format the input was delivered in. Unfortunately, this could not be done without much loss of structure. The explanation for this is that the trace of an average Asbru protocol involves lots of jumps from one branch to another. This is obvious, because an average guideline contains several parallel and cyclical plans. The consequence is that the trace simply does not have the form of a tree. It is a list of actions, and nothing more. Subsequently, one could then consider producing just a list in XML, but this would not be any improvement compared to the free text output. One could also consider producing an XML-tree only for those protocols that do not contain any parallel and cyclical plans, but since these protocols hardly exist, this would be a useless extension. Hence, the plan of producing the output in XML has finally been shelved completely.

Another option is not to produce an XML-file, but an HTML file. Nevertheless, this option would not bring much useful functionality as well. It might arrange the output a little bit better, but the general idea would still remain the same. Thus, since it is highly questionable whether this idea would be worth the efforts, it has not been realised as yet. Actually, both options mentioned would be better realised as a post-processing by a separate tool, rather than folding the functionality into our tool.

A last option that would potentially improve the utility of the output, is to print only the state transitions of the atomic plans. In other words, when the hierarchy of the plans would be written in the form of a tree, this would mean that only the leaves would be printed. And this certainly is a useful thing, since it would provide a clear image of the order in which all leaves are executed, without having any noise caused by the parent plans. In the current version of the interpreter, this option has not been realised yet. Hence, at the moment the output is still in free text. However, for future work it would be a simple intervention to implement the output just discussed, namely by adding a rule that would check if the plan is actually atomic, before writing it to the output.

6.3 Example output

In the last three sections, we have tried to give an image of how the code both the Prolog and Clips program has been arranged. In order to illustrate how the programs work, this section will give examples of what output the programs will produce, given certain input. The first subsection will deal with the Prolog part, the second one with the Clips part.

6.3.1 The Prolog parser

Figures 35, 36 and 37 provide the exact sets of facts the Prolog program will generate when offering it part of the Jaundice protocol, namely the plans Hyperbilirubinemia, Check-for-jaundice-after-2-weeks, and Diagnostics-and-Treatment-hyperbilirubinemia (see respectively figure 8, 9 and 10) as input. Note that this is not a realistic combination of plans, but for this example they largely satisfy.

When one takes a look at the figure 35, one discovers that the first fact that has been produced is the "start" fact, needed to indicate which plan is the top level plan. Next, the toplevel plan itself has been created as a proper plan with four 'unordered' subplans. An atomic abort condition has been created as well.

(start Hyperbilirubinemia)

(plan (name Hyperbilirubinemia) (proper yes) (abort_cond abort_Hyperbilirubinemia) (type do_unordered) (wait_optional yes) (cont_spec Diagnostics-and-Treatment-hyperbilirubinemia) (subplans Check-for-rapid-TSB-increase Check-for-jaundice-after-2-weeks Check-for-jaundice-after-3-weeks Diagnostics-and-Treatment-hyperbilirubinemia))

(exp (name abort_Hyperbilirubinemia) (operator atomic) (type cond) (subexps possibility-of-hemolytic-disease equal yes))

Figure 35. Set of Clips facts representing the plan Hyperbilirubinemia.

In figure 36, there are three proper plans (lines 1-6), two of which are user performed plans. These two plans have been created while parsing the Check-for-jaundice-after-2-weeks plan (see figure 9). Furthermore, the body of this plan has been decomposed into two new plans, one of type any-order (line 8), and one of type if-then-else (line 11). After that, a list of fictive plans of all different types can be seen (lines 14-30). Remarkable is that, whenever an if-then-else plan occurs, the sets of plans that should be executed in both the then-branch and the else-branch, have automatically been grouped into a sequential plan. Finally, the set of expressions are shown (lines 32-42), of which one has type or, and the rest is atomic. Note that they all have type *cond*, because in this example, all continuation specifications will be generated during the pre-processing of the Clips program. Also note that some of the expressions are conditions, others are if-branches for if-then-else plans.

```

1 (plan (name Check-for-jaundice-after-2-weeks) (proper yes) (filter_cond filter_Check-for-jaundice-after-2-weeks) (type do_sequentially)
(cont_spec all) (subplans Check-for-jaundice-after-2-weeks_1 Check-for-jaundice-after-2-weeks_2))

(plan (name Exit-possibility-of-cholestatic-jaundice) (proper yes) (type user_performed))
5
(plan (name Exit-provide-routine-care) (proper yes) (type user_performed))

(plan (name Check-for-jaundice-after-2-weeks_1) (type do_any_order) (cont_spec all) (subplans Check-for-jaundice-after-2-weeks_1_1
Check-for-jaundice-after-2-weeks_1_2 Check-for-jaundice-after-2-weeks_1_3))
10
(plan (name Check-for-jaundice-after-2-weeks_2) (type if_then_else) (subplans Check-for-jaundice-after-2-weeks_2_if Check-for-jaundice-
after-2-weeks_2_then Check-for-jaundice-after-2-weeks_2_else))

(plan (name Check-for-jaundice-after-2-weeks_1_1) (type ask) (subplans physical-exam-OK))
15
(plan (name Check-for-jaundice-after-2-weeks_1_2) (type ask) (subplans colour-stools))

(plan (name Check-for-jaundice-after-2-weeks_1_3) (type ask) (subplans colour-urine))

20 (plan (name Check-for-jaundice-after-2-weeks_2_then) (type do_sequentially) (subplans Check-for-jaundice-after-2-weeks_2_then_1
Check-for-jaundice-after-2-weeks_2_then_2 Exit-possibility-of-cholestatic-jaundice))

(plan (name Check-for-jaundice-after-2-weeks_2_else) (type do_sequentially) (subplans Check-for-jaundice-after-2-weeks_2_else_1 Exit-
provide-routine-care))
25
(plan (name Check-for-jaundice-after-2-weeks_2_then_1) (type ask) (subplans direct-serum-bilirubin))

(plan (name Check-for-jaundice-after-2-weeks_2_then_2) (type assignment) (subplans possibility-of-cholestatic-disease yes))

30 (plan (name Check-for-jaundice-after-2-weeks_2_else_1) (type assignment) (subplans possibility-of-cholestatic-disease no))

(exp (name Check-for-jaundice-after-2-weeks_2_if) (operator or) (type cond) (subexps Check-for-jaundice-after-2-weeks_2_if_1 unknown
Check-for-jaundice-after-2-weeks_2_if_2 unknown Check-for-jaundice-after-2-weeks_2_if_3 unknown))

35 (exp (name Check-for-jaundice-after-2-weeks_2_if_1) (operator atomic) (type cond) (subexps physical-exam-OK equal no))

(exp (name Check-for-jaundice-after-2-weeks_2_if_2) (operator atomic) (type cond) (subexps colour-stools equal "light"))

(exp (name Check-for-jaundice-after-2-weeks_2_if_3) (operator atomic) (type cond) (subexps colour-urine equal "dark"))
40
(exp (name filter_Check-for-jaundice-after-2-weeks) (operator atomic) (type cond) (subexps jaundice-clinically-significant equal yes)
(period 2 w 2 w) (since start))

```

Figure 36. Set of Clips facts representing the plan Check-for-jaundice-after-2-weeks.

Figure 37 contains the sequential plan Diagnostics-and-Treatment-hyperbilirubinemia (line 1), and two subplans asking for values (lines 6 and 8). Just like in the previous figure, many kinds of expressions can be seen (lines 10-27), which we will not explain in detail.

```

1 (plan (name Diagnostics-and-Treatment-hyperbilirubinemia) (proper yes) (complete_cond complete_Diagnostics-and-Treatment-
hyperbilirubinemia) (abort_cond abort_Diagnostics-and-Treatment-hyperbilirubinemia) (type do_sequentially) (cont_spec none) (subplans
Diagnostics-and-Treatment-hyperbilirubinemia_1 Diagnostics-and-Treatment-hyperbilirubinemia_2 Diagnostics-hyperbilirubinemia
Treatment-hyperbilirubinemia))
5
(plan (name Diagnostics-and-Treatment-hyperbilirubinemia_1) (type ask) (subplans term-child))

(plan (name Diagnostics-and-Treatment-hyperbilirubinemia_2) (type ask) (subplans age-child))

10 (exp (name complete_Diagnostics-and-Treatment-hyperbilirubinemia) (operator or) (type cond) (subexps complete_Diagnostics-and-
Treatment-hyperbilirubinemia_1 unknown complete_Diagnostics-and-Treatment-hyperbilirubinemia_2 unknown))

```

```
(exp (name complete_Diagnostics-and-Treatment-hyperbilirubinemia_1) (operator atomic) (type cond) (subexps jaundice-clinically-
significant equal no))
15
(exp (name complete_Diagnostics-and-Treatment-hyperbilirubinemia_2) (operator atomic) (type cond) (subexps Treatment-
hyperbilirubinemia completed))

(exp (name abort_Diagnostics-and-Treatment-hyperbilirubinemia) (operator or) (type cond) (subexps abort_Diagnostics-and-Treatment-
20 hyperbilirubinemia_1 unknown abort_Diagnostics-and-Treatment-hyperbilirubinemia_2 unknown abort_Diagnostics-and-Treatment-
hyperbilirubinemia_3 unknown))

(exp (name abort_Diagnostics-and-Treatment-hyperbilirubinemia_1) (operator atomic) (type cond) (subexps term-child equal no))

25 (exp (name abort_Diagnostics-and-Treatment-hyperbilirubinemia_2) (operator atomic) (type cond) (subexps age equal day1))

(exp (name abort_Diagnostics-and-Treatment-hyperbilirubinemia_3) (operator atomic) (type cond) (subexps pathologic-reason equal yes))
```

Figure 37. Set of Clips facts representing the plans Hyperbilirubinemia, Check-for-jaundice-after-2-weeks, and Diagnostics-and-Treatment-hyperbilirubinemia.

6.3.2 The Clips program

Obviously, the set of facts shown in the above subsection is just a small subset of the set of facts that would be obtained when translating the complete Jaundice protocol to Clips facts. In order to test the interpreter, we have already run the program with the complete set of facts many times, with many different kinds of user input. These simulations result in very long traces of actions and user interactions. Part of such a trace is shown in figure 38. As one can see, in the beginning there is only one plan active. This is the top level plan Hyperbilirubinemia, which switches first to state considered, next to state ready, and then to state activated. As soon as it is activated, all its four subplans are being considered in an "unordered" way. Clips chooses for the most normal order, i.e. from left to right. Afterwards, the subplans' filter conditions are checked. Since only one of them is fulfilled, the corresponding subplan, Diagnostics-and-Treatment-hyperbilirubinemia, switches to state ready. The other subplans remain in state considered, according to the definition of the unordered plan. Note that, if the top level plan would have been sequential, the other subplans would already have been rejected. Next, Diagnostics-and-Treatment-hyperbilirubinemia switches to state activated without any problems. Right after that, its subplans are started in a sequential way. Its first subplan asks the user to enter whether the patient is a term-child. In this example, our answer to that question was affirmative. As a consequence, the value is stored (not visible) and the subplans immediately complete. At that moment, the second subplan can be activated, asking the user to enter the age of the child. When that has been done as well, Diagnostics-and-Treatment-hyperbilirubinemia can continue with its third subplan, being the proper plan Diagnostics-Hyperbilirubinemia. This course of things continues like this for a long time, until there are no more plans that can do a step. In the ideal case, this would be when the top level plan Hyperbilirubinemia completes as a result of a fulfilled continuation specification.

```
Hyperbilirubinemia considered
Hyperbilirubinemia ready
Hyperbilirubinemia activated
Check-for-rapid-TSB-increase considered
Check-for-jaundice-after-2-weeks considered
Check-for-jaundice-after-3-weeks considered
Diagnostics-and-Treatment-hyperbilirubinemia considered
Diagnostics-and-Treatment-hyperbilirubinemia ready
Diagnostics-and-Treatment-hyperbilirubinemia activated
Diagnostics-and-Treatment-hyperbilirubinemia_1 'activated'
Diagnostics-and-Treatment-hyperbilirubinemia_1 => term-child : Enter the value: yes
Diagnostics-and-Treatment-hyperbilirubinemia_1 completed
Diagnostics-and-Treatment-hyperbilirubinemia_2 'activated'
Diagnostics-and-Treatment-hyperbilirubinemia_2 => age-child : Enter the value: 30
Diagnostics-and-Treatment-hyperbilirubinemia_2 completed
Diagnostics-Hyperbilirubinemia considered
...and so on...
```

Figure 38. Part of output trace of the Clips program when simulating the Jaundice protocol.

The test case that provided the above trace was not the only test case we have run. On the contrary, we have tested the interpreter many times, thereby especially testing the most critical parts of the program, such as the rules that take care of the parallel behaviour. Also, we have tried to simulate some special cases, e.g. by entering such input that plans are immediately aborted or completed. In order to illustrate the interpreter's ability to deal with difficult situations, figure 39 shows a more complex trace. This part of the trace was obtained while the interpreter was just simulating the plan *Treatment-hyperbilirubinemia* (see figure 12). This plan is especially complex since it has a parallel plan body, with an any-order and a cyclical subplan. The first step the trace shows is the switch to state activated by the *Observation* plan (which is a subplan of *Regular-treatments*, see figure 12). The next step is a 'do nothing' step. This is because it is the turn of a subplan of the parallel plan *Treatment-hyperbilirubinemia*. However, this subplan cannot do any steps (because it is a cyclical plan, and the time point for the next cycle has not been reached yet). Hence, it is skipped. Subsequently, the other branch is allowed to do a step, which results in the activation of the *Prescribe-observation* plan. This parallel behaviour continues for a while, until the user is asked if the *Prescribe-observation* plan has completed. Since our answer to that question is no, the plan *Prescribe-observation* immediately aborts. By propagation, its parent *Observation* aborts too. At that moment a remarkable thing happens, namely the fact that the *Observation* plan is retried. The reason for this is that its parent *Regular-treatments* has the attribute *retry-aborted-children* set to *yes*. After the *Observation* plan switches to considered, the cyclical plan asking for values is finally allowed to do a step. Apparently, the time point for a new cycle has been reached. Here, we enter an extreme TSB-value of 50 in order to test the program. The result is astonishing. Since the previous TSB-value was much lower, the plan *Check-for-rapid-TSB-increase* is suddenly activated. Apparently, the filter condition of this plan, which is based on an abstraction of the TSB-value, has become true. And since its parent (*Hyperbilirubinemia*) is an unordered plan, it may be activated at any moment. The next steps demonstrate the activation of *Check-for-rapid-TSB-increase*. Finally, this plan discovers that there is a possibility of a hemolytic disease, and as a result, the top level plan is aborted.

```

...
Observation activated
    do nothing
Prescribe-observation considered
    do nothing
Prescribe-observation ready
    do nothing
Prescribe-observation activated
    do nothing
Prescribe-observation => Completed??? no
    Prescribe-observation aborted
    Observation aborted
    do nothing
retry Observation
    do nothing
Observation considered
Treatment-hyperbilirubinemia_2_1 (cycle)
Treatment-hyperbilirubinemia_2_1_1 'activated'
Treatment-hyperbilirubinemia_2_1_1 => TSB-value : Enter the value: 50
    Treatment-hyperbilirubinemia_2_1_1 completed
Check-for-rapid-TSB-increase ready
Check-for-rapid-TSB-increase activated
Check-for-rapid-TSB-increase_1 'activated'
Check-for-rapid-TSB-increase_1 => possibility-of-hemolytic-disease : yes
    Check-for-rapid-TSB-increase_1 completed
Hyperbilirubinemia aborted
...

```

Figure 39. Part of output trace of the Clips program when simulating the Jaundice protocol.

7. Evaluation

The goal of this chapter is to look back at the final behaviour of the interpreter, and to decide whether it satisfies the intended behaviour. When doing this, we will also name some experiences we had while experimenting with the Jaundice protocol. Furthermore, we will look back at the design choices made, and decide which of them we would do differently if we were to do the project over again.

When evaluating the final behaviour of the interpreter, one could conclude that it has become a very useful piece of software. The initial goal of the project, "writing an interpreter for the simulation of the application of clinical guidelines, written in the Asbru language", has been reached to a certain extent. Of course, it only supports a small part of Asbru, namely Asbru Light. And even this subset has not been covered completely. For instance, the current version of the interpreter is not able to deal with complex list manipulations, nor does it support the whole functionality of the cyclical plan, as defined in the reference manual. However, considering the time we had for this project and the enormous expressiveness of the Asbru language, these limitations are logical. It is better to have a small program that works correctly, than a big program full of bugs. As for our interpreter, we have already made clear that we ran many different tests in order to prove its correctness. Unfortunately, these tests were all based on a single protocol, so we cannot be 100% sure that the performance would be the same for another protocol. However, we can be sure of the fact that our program is able to simulate the Jaundice protocol and that is useful for the intended user group.

This last assertion can be underlined by means of some experiences we have had during testing. For example, in an earlier version of the Jaundice protocol, its top level was of type parallel instead of unordered. However, tests with that version demonstrated that the program then never had a chance to activate the plans Check-for-rapid-TSB-increase, Check-for-jaundice-after-2-weeks, and Check-for-jaundice-after-3-weeks, since they would be rejected almost at once. Obviously, this behaviour was in conflict with the designer's intentions, so that the plan was modified in a later version. Another minor error that the interpreter has detected was in a plan called *anamnesis-abnormal-signs*. Tests pointed out that some subplans of this plan were never executed, in situations where they should. The reason for this was that the plan's abort condition was mostly fulfilled too early. All these examples make clear that the interpreter can certainly be of some use, because it can help detecting errors that humans alone would not find.

Despite all of the above, there were some design choices that we would not make in the future. For instance, the way the priorities of the rules are determined, by means of the *salience* statements, is a bit confusing. It would be a challenge to see if the interpreter could be modified by leaving out these statements. Perhaps the ordering of the rules could be arranged by dividing the program into different modules. For instance, by having different modules for the state transitions, the evaluation of expressions and the execution of plans. Next to that, another thing that could be considered a limitation is that some parts of the programs have a rather complex structure. This is especially the case for the Prolog program, which contains some very long parsing predicates. Furthermore, one important condition that the designers of Asbru plans should have in mind when using the interpreter, is that two different plans are not allowed to have the same subplan (at least in the current version of the interpreter). This is because the status of a plan varies over time, so it is impossible that a single subplan is being modified because of two different reasons. Imagine for instance a subplan that has been started by its parent some steps ago, so that it is in state *ready*. Then, if another parent would suddenly start the same subplan, it would get back to state *considered*, which would interfere with the actions of the other parent plan. A solution for this problem is that, whenever two different plans need the same subplan, the designer should simply copy it and give the copy a new name. However, if we were to do the project over, it would be an option to represent the initial plans and the plans under execution by two different structures. This approach would solve the above problem. For the rest, most limitations in the current interpreter are just a matter of bad style, caused by a pressing deadline for the project. For instance, it would have been nicer to create a special slot for the queue of subplans of a parallel plan, instead of using the *subplans* slot.

The format in which the interpreter delivers its output is rather satisfactory. We do not think that the steps printed are too detailed. The fact that each single action or state transition is shown implies that even small errors in the behaviour of the protocol can be detected. These errors will mostly involve an incorrect order of execution, or plans that are not reached at all. However, one aspect of the output that might cause confusion to the user is the presence of fictive plans. Since these plans do not exactly correspond to the plans in the original Asbru protocols, reading the output trace will be a bit more difficult.

As for the choice for the programming language, on the whole, both Prolog and Clips were satisfactory for the intended purpose. The Prolog mechanism for parsing XML files was particularly useful, as well as Clips' ability

of checking for applicable rules. A minor drawback of Clips was that there did not exist a mechanism for evaluating expressions, so that we had to write all these rules by hand.

Summarising, we have constructed a useful interpreter by combining the Prolog mechanism for parsing XML files with Clips' ability of checking for applicable rules. Within the parsing part, nested Asbru plans are divided into several fictive plans, all represented with the template *plan*. The advantage of this approach is that the resulting plans are less complex. However, it makes the final output a bit harder to understand, since it does not correspond exactly to the original Asbru protocol anymore. Furthermore, the different Asbru state transitions are implemented by Clips rules checking for the applicability of conditions. These conditions are represented with the template *exp*, which is comparable to the *plan* template. It was designed in such a way that all expressions can continuously be evaluated in several steps. This makes that state transitions may occur at any desired moment. As for the execution of plans, all Asbru Light control structures, of which the parallel plan is the most complex, have been implemented by means of Clips rules as well. The fact that plans may activate their children within these rules makes that all possible combinations of control structures can be executed. The result of all this is a program that performs well with the simulation of the Jaundice protocol. It passed a large set of tests for special cases, and has demonstrated its benefit by detecting some small errors in the Asbru Jaundice protocol. We cannot say how well it will perform with other protocols. The limitations it has are mostly matters of programming style. They are not really disturbing for the users.

8. Future work

Future steps with respect to this project would involve the addition of more Asbru concepts in the first place. We have already demonstrated that, after some rearrangements, the interpreter could relatively easily be extended, e.g. by adding more list manipulations, or by extending the possibilities of the cyclical plan. Additionally, the support of time annotations could be completed, and one could even try to add the notion of preferences and effects. Besides this, we think that another important future goal might be to test the interpreter on other guidelines than the Jaundice protocol. Doing this would be profitable for both sides, because it could help to discover errors in both the guideline and the interpreter.

Furthermore, it might even be possible to use the interpreter in the future for other purposes than just designing protocols. One could for instance think of the purpose of critiquing. At the moment, all that the program does with intentions is to parse them away. However, an option is to store the intentions as well, just as we do with conditions. This way, we could have the interpreter print them any time that the corresponding plan is used. This approach would allow the researchers to match them to the intentions used by an expert in a very detailed way, so that it could contribute to the field of critiquing.

Appendix A: Introduction to Prolog and Clips

Introduction to Prolog

A predicate in Prolog is called a *clause*. Prolog clauses usually consist of a *head* and a *body*. A clause whose head is empty is called a *fact*. Consider the following set of facts:

```
male(john).
female(mary).
male(bill).
parent(john, bill).
parent(mary, bill).
```

Prolog has a searching mechanism that can find all solutions to a certain *query*, a question by the user. Suppose for instance we would enter the query “male(X)”. Given the above set of facts, Prolog would answer that both John and Bill are solutions. With this mechanism in mind, more complex clauses can be built. For instance, a clause defining the ‘father’ relationship would look like this:

```
father(Father, Child):-
    male(Father),
    parent(Father, Child).
```

In this example, the statement before the “:-” is the head, the rest is the body. Note that variables are always written with a capital. For a certain variable, the head of a clause is true if all statements in the body are true. Hence, if we would enter the query “father(john, bill)”, the program would answer that this is true. Likewise, if we would enter the query “father(john, X)”, the program would answer that Bill is the only solution.

A Prolog program consists of a hierarchy of clauses. This means that clauses may occur within the body of other clauses. The program is executed by prompting the top level clause.

Introduction to Clips

A Clips program is based on a set of facts that hold in a certain state of the world, combined with a set of rules that can fire in certain situations. It consists of the following elements:

- *deftemplates*: Define the names and types of the different attributes a fact may have, by means of *slots*. For instance, a deftemplate for a person might contain a slot with name “name” and type “symbol”, as shown in the example below. Many additional settings like a restricted set of allowed symbols or a default value may be defined as well. Furthermore, attributes may also be defined by means of a *multislot* instead of a slot, meaning that more than one value is allowed.

```
(deftemplate person
```

```
    (slot name
      (type SYMBOL))
    (slot age
      (type NUMBER))
    (multislot names_children
      (type SYMBOL)
      (default none))
```

```
)
```

- *deffacts*: Hold in certain world states. Some of them are defined within the program, which means that they always hold. Others may be asserted later. Existing facts can be modified and retracted at any time. Facts are written down in the following syntax:

```
(deffacts family_members
```

```
  (person (name john) (age 40) (names_children bill lucy))
  (person (name mary) (age 38) (names_children bill lucy))
  (person (name bill) (age 12) (names_children none))
  (person (name lucy) (age 8) (names_children none))
)
```

- *deffunctions*: These are functions comparable to those in other programming languages (e.g. to evaluate if a list has an even number of arguments). They are activated by *defrules*.
- *defrules*: Consist of an if-part and a then-part. The if-part is a set of statements that check if certain conditions, mostly based on existing facts, hold. This checking is done continuously from the moment that the program is started. As soon as the whole if-part of a rule holds, the then-part is being executed. An example of a *defrule* is shown below. Notice that the if-part and the then-part are separated by a “=>”. Names that start with a “?” are variables.

```
(defrule print_age
```

```
  (person (name ?name) (age ?age))
  =>
  (printout t ?name)
  (printout t "s age is: ")
  (printout t ?age)
)
```

In addition to the kind of statements shown above, the if-part can be provided with a *salience* statement. This is a special construct defining a number that indicates the priority of the rule. This is needed because there are situations when several rules can fire at the same time, and executing them in different orders also means different behaviour. In these situations Clips always picks the rule with the highest salience number. The default salience value of each rule is 0.

A Clips program is defined simply by defining all elements just mentioned. It can be started by typing “(run)”. This means that the loop, checking for the applicability of rules and executing the applicable ones, is started. The program terminates as soon as there are no more applicable rules.

Appendix B: Main deftemplates and defrules of the Clips program

Deftemplate for representing plans

```
(deftemplate plan

  "An Asbru plan. Examples:
  (plan (name plan1) (proper yes) (abort_cond cond1) (type do_unordered)
        (wait_optional yes) (cont_spec spec1) (subplans plan2 plan3 plan4))
  (plan (name plan2) (type user_performed))"

  (slot name
    (type SYMBOL))
  (slot type
    (type SYMBOL)
    (allowed-symbols do_unordered do_sequentially do_any_order
                      do_parallel cyclical if_then_else ask assignment manipulation      display
                      user_performed do_for_each))
  (slot proper
    (type SYMBOL)
    (allowed-symbols yes no)
    (default no))
  (multislot subplans)
  (slot cont_spec
    (type SYMBOL))
  (slot retry_aborted
    (type SYMBOL)
    (allowed-symbols yes no)
    (default no))
  (slot wait_optional
    (type SYMBOL)
    (allowed-symbols yes no)
    (default no))
  (slot activation
    (type SYMBOL)
    (allowed-symbols automatic manual)
    (default automatic))
  (slot filter_cond
    (type SYMBOL))
  (slot abort_cond
    (type SYMBOL))
  (slot complete_cond
    (type SYMBOL))
  (multislot input_args
    (type SYMBOL))
  (multislot output_args
    (type SYMBOL))
  (multislot repeat)
  (slot state
    (type SYMBOL)
    (allowed-symbols ignored considered ready activated aborted completed
                      rejected)
    (default ignored))
  (multislot invoked_by
    (type SYMBOL)
    (default none))
  (slot your_turn
    (type SYMBOL)
    (allowed-symbols yes no)
    (default no))
  (slot since
    (type NUMBER))
)
```

Defrules for state transitions

```
(defrule consider_plan_1
```

```
"The plan is a proper plan: transition from ignored to considered.  
If plan is a child of a parallel plan, schedule the invokers."
```

```
  (declare (salience 5))  
  ?f <- (plan (name ?name) (type ?type) (proper yes) (state ignored)  
    (invoked_by $?invokers) (your_turn yes))  
  (start ?name)  
  =>  
  (printout t ?name)  
  (printout t " considered" crlf)  
  (modify ?f (state considered))  
  (assert (schedule $?invokers))  
)
```

```
(defrule consider_plan_2
```

```
"The plan is a subplan: transition from ignored to 'activated'."
```

```
  (declare (salience 5))  
  ?f <- (plan (name ?name) (type ?type) (proper no) (state ignored)  
    (invoked_by ?invokers) (your_turn yes))  
  (start ?name)  
  =>  
  (printout t ?name)  
  (printout t " 'activated'" crlf)  
  (modify ?f (state activated) (since (time)))  
  (assert (schedule $?invokers))  
)
```

```
(defrule filter_condition_1
```

```
"Transition from considered to ready.  
No filter condition."
```

```
  (declare (salience 5))  
  ?f <- (plan (name ?name) (filter_cond nil) (state considered)  
    (invoked_by $?invokers) (your_turn yes))  
  =>  
  (printout t ?name)  
  (printout t " ready" crlf)  
  (modify ?f (state ready))  
  (assert (schedule $?invokers))  
)
```

```
(defrule filter_condition_2
```

```
"Transition from considered to ready.  
Filter condition is true."
```

```
  (declare (salience 5))  
  ?f <- (plan (name ?name) (filter_cond ?fc) (state considered)  
    (invoked_by $?invokers) (your_turn yes))  
  (exp (name ?fc) (value true) (explanation ?text))  
  =>  
  (printout t ?name)  
  (printout t " ready" crlf)  
  (printout t "Explanation: ")  
  (printout t ?text crlf)  
  (modify ?f (state ready))  
  (assert (schedule $?invokers))  
)
```

```

(defrule filter_condition_3a

"Transition from considered or ready to rejected.
Plan has no parents.
Filter condition is not true.
Note that priority is lower than those of skip_2a & skip_2b!"

  (declare (salience 60))
  ?f <- (plan (name ?name) (filter_cond ?fc) (state ?state) (invoked_by $?invokers)
            (your_turn yes))
  (not (plan (subplans $? ?name $?) (type ?type)))
  (test (neq ?fc nil))
  (not (exp (name ?fc) (value true)))
  (test (or (eq ?state considered) (eq ?state ready)))
  =>
  (printout t ?name)
  (printout t " rejected" crlf)
  (modify ?f (state rejected))
  (assert (schedule $?invokers))
)

(defrule filter_condition_3b

"Transition from considered or ready to rejected.
Plan has a non-unordered parent.
Filter condition is not true.
Note that priority is lower than those of skip_2a & skip_2b!"

  (declare (salience 60))
  ?f <- (plan (name ?name) (filter_cond ?fc) (state ?state) (invoked_by $?invokers)
            (your_turn yes))
  (plan (subplans $? ?name $?) (type ?type))
  (test (neq ?type do_unordered))
  (test (neq ?fc nil))
  (not (exp (name ?fc) (value true)))
  (test (or (eq ?state considered) (eq ?state ready)))
  =>
  (printout t ?name)
  (printout t " rejected" crlf)
  (modify ?f (state rejected))
  (assert (schedule $?invokers))
)

(defrule filter_condition_3c

"Transition from considered or ready to rejected.
Plan has an unordered parent.
Filter condition is not true.
Note that the priority is lower than all others, since the rule
must wait until other subplans are done!"

  (declare (salience -600))
  ?f <- (plan (name ?name) (filter_cond ?fc) (state ?state) (invoked_by $?invokers)
            (your_turn yes))
  (plan (subplans $? ?name $?) (type do_unordered) (state activated))
  (test (neq ?fc nil))
  (not (exp (name ?fc) (value true)))
  (test (or (eq ?state considered) (eq ?state ready)))
  =>
  (printout t ?name)
  (printout t " rejected" crlf)
  (modify ?f (state rejected))
  (assert (schedule $?invokers))
)

```

```

(defrule automatic_start_condition

"Transition from ready to activated.
Activation is automatic.
If plan is subplan of a parallel plan, all other subplans must be at least
ready to go to the next state."

(declare (salience 50))
?f <- (plan (name ?name) (activation automatic) (state ready)
(invoked_by $?invokers) (your_turn yes))
(not (and (plan (type do_parallel) (subplans $?subplans)) (plan (name ?name2)
(state considered)) (test (member$ ?name $?subplans))
(test (member$ ?name2 $?subplans)) ))
; There is no parallel plan, of which both ?name and a considered
plan are a subplan
=>
(printout t ?name)
(printout t " activated" crlf)
(modify ?f (state activated) (since (time)))
(assert (schedule $?invokers))
)

(defrule manual_start_condition_1

"Ask if plan is allowed to start.
Activation is manual."

(declare (salience 50))
?f <- (plan (name ?name) (activation manual) (state ready) (your_turn
yes))
(not (and (plan (type do_parallel) (subplans $?subplans)) (plan (name ?name2)
(state considered)) (test (member$ ?name $?subplans))
(test (member$ ?name2 $?subplans)) ))
; There is no parallel plan, of which both ?name and a considered
plan are a subplan
(not (manual_start ?name ?))
=>
(printout t "=> ")
(printout t ?name)
(printout t " : ")
(ask_start ?name)
)

(defrule manual_start_condition_2a

"If plan is allowed to start, transition from ready to activated"

(declare (salience 50))
?f <- (plan (name ?name) (activation manual) (state ready)
(invoked_by $?invokers) (your_turn yes))
(manual_start ?name yes)
=>
(printout t ?name)
(printout t " activated" crlf)
(modify ?f (state activated) (since (time)))
(assert (schedule $?invokers))
)

(defrule manual_start_condition_2b

"If plan is not allowed to start, transition from ready to rejected"

(declare (salience 50))
?f <- (plan (name ?name) (activation manual) (state ready)
(invoked_by $?invokers) (your_turn yes))
(manual_start ?name no)
=>
(printout t ?name)
(printout t " rejected" crlf)
(modify ?f (state rejected))
(assert (schedule $?invokers))
)

```

```

(defrule abort_condition

"Transition from activated to aborted.
Abort condition is true."

  (declare (salience 100))
  ?f <- (plan (name ?name) (abort_cond ?ac) (state activated)
    (invoked_by $?invokers) (your_turn yes))
  (exp (name ?ac) (value true) (explanation ?text))
  =>
  (printout t ?name)
  (printout t " aborted" crlf)
  (printout t "Explanation: ")
  (printout t ?text crlf)
  (modify ?f (state aborted))
  (assert (schedule $?invokers))
)

(defrule retry_aborted_children

"Transition of a child from aborted to ignored.
Retry_aborted of parent is yes.
Parent is still active."

  (declare (salience -100))
  ?f <- (plan (retry_aborted yes) (subplans $? ?name $?) (state ?state))
  (test (and (neq ?state completed) (neq ?state aborted)))
  (plan (name ?name) (state aborted) (invoked_by $?invokers) (your_turn yes))
  =>
  (printout t "retry ")
  (printout t ?name crlf)
  (assert (reset ?name))
  (assert (schedule $?invokers))
  ;***retry them over and over***
)

```

Bibliography

- [1] Hibble A., Kanka D., Pencheon D., Pooles F. – *Guidelines in general practice: the new tower of Babel?* British Medical Journal vol. 317, 26-09-1998, pp. 862-863.
- [2] Lobach D.F., Hammond E. – *Computerized Decision Support Based on a Clinical Practice Guideline Improves Compliance with Care Standard* American Journal of Medicine vol. 102, january 1997, pp. 89-97.
- [3] Pisanelli D.M., Gangemi A., Steve G. – *The Role of Ontologies for an Effective and Unambiguous Dissemination of Clinical Guidelines* Juan les Pins, France: 12th international conference on knowledge engineering and knowledge management, 2000.
- [4] American Society of Internal Medicine – *Using Practice Guideline Compendiums To Provide Better Preventive Care* Annals of Internal Medicine vol. 130, 02-03-1999, pp. 454-457.
- [5] Shahar Y., Miksch S., Johnson P. – *The Asgaard Project: A Task-Specific Framework for the Application and Critiquing of Time-Oriented Clinical Guidelines* Artificial Intelligence in Medicine, 14, 1998, pp. 29-51.
- [6] Ohno-Machado L., Gennari J.H., Murphy S.N., Jain N.L., Tu S.W., Oliver D.E., Pattison-Gordon E., Greenes R.A., Shortliffe E.H., Octo Barnett G. – *The Guideline Interchange Format: a model for representing guidelines* JAMIA 1998, 5, pp. 357-372.
- [7] Vollebregt A., ten Teije A., van Harmelen F., van der Lei J., Mosseveld M. – *A study of PROforma, a development methodology for clinical procedures* Artificial Intelligence in Medicine, 1999.
- [8] Field M., Lohr K. – *Clinical Practice guidelines: Directions for a New Program* Institute of Medicine, Washington DC, National Academy Press, 1990.
- [9] Roomans H.F. – *Formalisation of medical guidelines* Vrije Universiteit Amsterdam, august 2000.
- [10] Woolf S.H., Grol R., Hutchinson A., Eccles M., Grimshaw J. – *Potential benefits, limitations, and harms of clinical guidelines.* British Medical Journal vol. 318, 20-02-1999, pp. 527-530.
- [11] Miksch S., Seyfang A. – *Continual Planning with Time-Oriented Skeletal Plans* ECAI 2000.
- [12] Miksch S. – *Plan Management in the Medical Domain* AI Communications, 4, 1999.
- [13] Kosara R., Miksch S. – *A User Interface for Executing Asbru Plans* Artificial Intelligence in Medicine, 2001, pp. 136-139.
- [14] Seyfang A., Miksch S., Horn W., Urschitz M.S., Popow C., Poets C.F. – *Using Time-Oriented Data Abstraction Methods to Optimize Oxygen Supply for Neonates* Artificial Intelligence in Medicine, 2001, pp. 217-226.
- [15] Miksch S., Kosara R., Shahar Y., Johnson P. – *AsbruView: Visualisation of Time-Oriented Skeletal Plans* Menlo Park, CA: 4th international conference on AI planning systems, 1998.
- [16] Marcos M., Berger G., van Harmelen F., ten Teije A., Roomans H. – *Using critiquing for improving medical protocols: harder than it seems*

Artificial Intelligence in Medicine, 2001, pp. 431-441.

- [17] Duftschmid G., Miksch S., Shahar Y., Johnson P. – *Multi-Level Verification of Clinical Protocols*
Trento, Italy: 6th international conference on principles of knowledge representation and reasoning, 1998.
- [18] Balser M., Reif W. – *KIV research*
<http://www.informatik.uni-augsburg.de/swt/fmg/>
- [19] Seyfang A., Kosara R., Miksch S. – *Asbru 7.2 reference manual*
University of Vienna, 22-12-2000.
- [20] Miksch S., Shahar Y., Johnson P. – *Asbru: a Task-Specific, Intention-Based, and Time-Oriented Language for Representing Skeletal Plans*
Keynes, UK: 7th workshop on knowledge engineering: methods & languages, 1997.
- [21] Friedland P.E., Iwasaki Y. – *The Concept and Implementation of Skeletal Plans*
Journal of Automated Reasoning 1985, 1, pp. 161-208.
- [22] Marcos M., van Harmelen F., ten Teije A. – *Asbru protocol for the Management of Hyperbilirubinemia in the Healthy Term New-born* (working document)
Vrije Universiteit Amsterdam, 2001.
- [23] American Academy of Pediatrics – *Management of Hyperbilirubinemia in the Healthy Term New-born*
Pediatrics 1994, vol 94(4).
- [24] Marcos M., van Harmelen F., ten Teije A. – *Asbru protocol for the Management of Diabetes Mellitus Type 2* (working document)
Vrije Universiteit Amsterdam, 2001.
- [25] Wielemaker J. - *SWI-Prolog version 4.0.0*
<http://www.swi.psy.uva.nl/projects/SWI-Prolog/>
- [26] *Clips basic reference manual*
http://www.cs.vu.nl/~ksprac/doc/CLIPS_Basic_Reference_Manual/top.html
- [27] Bratko I. – *Prolog Programming for Artificial Intelligence*
Addison-Wesley Publishers Ltd., 1990, pp. 4-27.