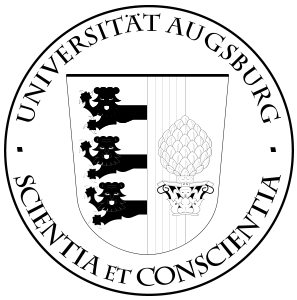
Universität Augsburg



Support for Interactive Verification of Asbru in KIV

J. Schmitt, M. Balser, W. Reif

Report 2006-16 June 2006

Institut für Informatik
D-86135 Augsburg

Abstract

This technical report documents the integration of the Asbru modelling language in the interactive verification environment KIV. Asbru is useful to model medical guidelines. KIV is a powerful tool for the application of formal methods. With our integration, it is possible to apply formal interactive verification to Asbru medical guidelines. [1]

[1] This work has been partially funded by the European Commission's IST program, under contract number IST-FP6-508794 Protocure II.

Contents

1 Introduction

This technical report documents the integration of Asbru in KIV.

Asbru [9, 5] is a plan-oriented language especially designed for the modelling of medical guidelines. A formal semantics of Asbru has been defined in [3, 7]. The semantics is a prerequisite for formal analysis. For interactive verification of Asbru medical guidelines, we have added support to the interactive verification environment KIV. KIV offers algebraic specifications and strong support for reasoning in higher order logic. Special support for temporal logic has recently been added. Asbru medical guidelines can be seen as a special form of reactive system. We have integrated Asbru in KIV such that temporal properties can be verified with the proof method of symbolic execution.

For the integration of Asbru in KIV, an algebraic specification for the syntax and semantics – as defined in [7] – of Asbru has been constructed. This document gives an overview of the specification and highlights important details. The KIV system can be obtained at [4].

Studies have already been conducted to evaluate the usefulness of this implementation with respect to the feasibility of the verification of Asbru properties. As the scope of this report is the implementation not the verification, detailed results regarding verification are not published within this document, but in [6]. The results obtained so far strongly indicate the usefulness of this implementation.

This report describes the implemented Asbru semantics version 2.12. This version of the Asbru semantics is based on the Asbru language version 7.3.

2 Development graph

In Fig. 1 the development graph of our implementation of the Asbru semantics is presented. The specifications with depicted cutters belong to the library specifications included in KIV. they will not be described within this paper.

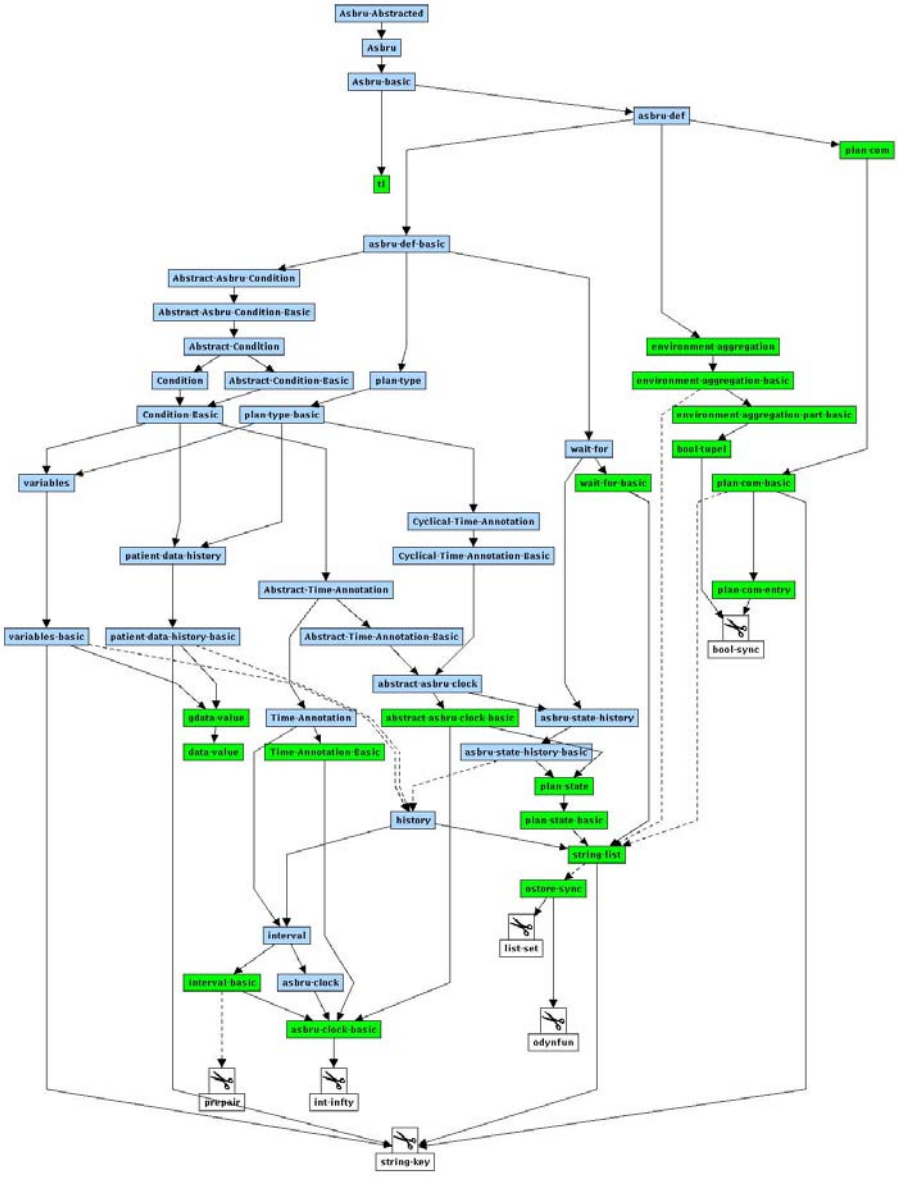


Figure 1: Asbru specification in KIV

3 Identifiers

Identifiers such as names of Asbru plans are modelled as strings in KIV.

3.1 string-key

The string-key specification is an enrichment of the KIV strings. It has been defined as a collection of further simplifier rules and to define necessary predicates and functions for the use of strings as unique keys. Currently it is an empty specification.

3.2 string-list

string-list is a refinement of lists with strings as elements. In general, string-lists represent lists of names of Asbru plans, e.g. as sub-plans for higher level plans.

4 Time

4.1 asbru-clock-basic

This specification defines the data type asbru-clock. The asbru-clock is a two component counter, with the first component being either an integer or infinity, the second component being a natural number.

The first component of the clock counts the time steps, the system has gone through. For reasons of modularisation, it has been decided, this counter should be an integer rather than a natural number. In general, the absolute tick number is considered to be unimportant. This way, lemmas can be inserted at different time points. It is not necessary to deal with the difficulty, that with a natural number as a tick counter there is some zero time point, so that you cannot go infinitely back into history. The tick counter may reach infinity or negative infinity when adding the offsets of time annotation time points, which in turn can be infinity or negative infinity.

The second counter is a micro-step counter. A micro-step in the context of Asbru is a technical step, where the environment is not allowed to change. As Asbru is a mainly control oriented language, many control steps are considered to take no real time at all. Examples for this are plan state changes. These can be handled by the system in virtually no time at all and therefore should not lead to a timely progress. Therefore, timing in Asbru is based on the strong synchronous hypothesis [1].

There are different possibilities as to how this micro and macro step scheme may be implemented, one being some form of consensus semantics, where each running plan has to agree on the fact, that no further micro-step is necessary. The second implementation is a more simple principle, where a fixed number of n steps are regarded as micro-steps, while only every (n + 1)-th step is a macro-step. This second concept has been rejected by the KIV Asbru semantics implementation, because of its theoretical limits. For example, it is in general

not known, that a certain amount of micro-steps is sufficient. It might be possible, that it is necessary to send signals several times through the whole plan hierarchy and the maximum size of this hierarchy is in general unknown.

4.2 asbru-clock

The specification asbru-clock enriches the asbru clock um mathematical functionality. The functions added here allow to move forward and backward in the time and to compare time-points. It furthermore defines abbreviations to hide the details of modifying the clock within micro and macro-steps by functions

micro-step : asbru-clock -> asbru-clock
macro-step : asbru-clock -> asbru-clock

Where the micro-step increases the micro-step counter and the macro-step increases the macro-step counter while resetting the micro-step counter.

In general, it has been the intention of the specification of the mathematics regarding asbru-clocks to express the fact, that micro-steps do not represent real time steps and therefore are not related to concepts as "earlier" or "later". It is therefore not possible to address individual micro-steps but only to refer to a list of states that has been reached in between two macro-steps.

Additionally, minimum and maximum functions for simple access to the earliest or latest time out of a pair of asbru-clocks are defined.

Currently only a partial mapping between the Asbru ticks and real world time is being defined. Current examples work well with an assumption, that a second, which is the smallest time unit currently used in the case studies, can be divided into n Asbru ticks, where n is an arbitrary but fixed value. Apart from that, the logical upscale has been specified, where 60 seconds are equal to one minute, 60 minutes to one hour and so on. If it is ever necessary to specify ticks further, one can do so in the case study dependent specifications.

5 Intervals

5.1 interval-basic

interval-basic is a pair of asbru-clocks and therefore a rudimentary time-interval.

5.2 interval

interval enriches interval-basic with simplifier rules and additional predicates. Especially the validity of an interval is defined within this specification. An interval is valid, if the lower border (i.e. the earlier time point) really is lower (or earlier) then the upper border. A time point is member of an interval, iff it is not earlier than the first time point and not later than the second. With this definition of the elem relation for times, an interval not satisfying the validity would not contain a single time point.

From a theoretical point of view, it is not feasible to specify that the first time

point must not be later than the second on a syntactical level, which would lead to an inconsistent specification. It would have been possible to specify the selector functions on intervals that way, so that one selector always returns the earlier time-point and one selector always the later time-point. However it has been decided not to do so. The reason for this is a minimation of the case distinctions within proofs. In general the knowledge about the time points of an interval is limited and sometimes it cannot be easily (i.e. automated) decided whether a time point is earlier than the other one or not. It has therefore been considered the duty of the verifying person to ensure he validity of intervals.

6 Dynamic Functions

6.1 ostore-sync

The ostore-sync specification enriches the specification dynfun with the specification of the predicate sync. The sync predicate is a way to come around difficulties with concurrent access to data types within synchronous parallel execution. In general synchronous write access from more than one process to one variable is seen as a clash. The result of such a clash can be defined differently. Some common specifications to deal with clashes are the following:

- The result of the assignment is arbitrary chosen from one of the assigning processes (one process overwrites the other ones result)
- The result is arbitrary (interleaved writing might lead to inconsistent data structures)
- Both assignments are carried out (i.e. access to different fields within an array)
- The assignment results in an inconsistency, the program "aborts"

In general it depends on the actual problem, which of these specifications best represents the desirable outcome. The sync predicate postpones the decision how to react to clashes and allows, it not to be specified in the KIV foundation but on case study level. So ostore-sync defines the outcome of concurrent access to dynamic functions.

A dynamic function can be seen as something like a hashtable. Synchronous access has been defined in a way that concurrent updates to different fields are executed without a clash, that is, both updates are executed. Conflicting updates result in an non solvable clash, that is, the program aborts with a finite error. Data types as used within this specification are not capable of detecting an update with the original data value. This could be an assignment of the form f(x) := x. Such an assignment would be considered not to have happened from the sync point of view. If this is considered as a shortcoming (which it is not for this project) it can be avoided by using boolean flags to indicate updates.

Dynamic functions as data structures represent a state of the system or the patient. Details about the usage of dynamic functions for data storage can be found within the description of the asbru-state and patient-data specifications.

In principle, sync predicates have to be defined for every specification, where synchronous parallel write access to one variable may occur. Such specifications are history, asbru-clock and further specifications.

In addition to the specification of sync also simplifier and forward rules are specified to automatically deal with sync predicates in proof obligations such that it can be determined automatically whether a clash has occurred and also, what the outcome of a parallel non-clashing update is.

7 Histories

7.1 history

The concept of the history data type is very close to the concept of dynamic functions. The key of the history selectors are time points. For reasons of modularisation it has been decided to add a possibility to work not on single time points but also on intervals.

History is a generic specification with the type of the included dynamic function left undefined. With this it is possible to define generic simplification rules. This saves the effort for reproving and specification for the actualised dynamic functions. The history construct as it is defined is used by three specifications, the variable history, the Asbru state history and the patient data history.

8 State

8.1 Important data structures

The most important data structures within Asbru are the Asbru State, Patient Data and Patient. Within the Asbru state all configurations of Asbru plans are stored, that is their current state regarding the semantics state-chart. Within the Patient Data known values of the patient are stored. It is important to distinguish between the Patient data structure and the Patient Data data structure. While the former contains information about the real condition of the patient, the latter represents the knowledge the medical staff has gained about the patient. This knowledge can be outdated in the sense, that values are already changed within the patient but not yet updated in the records. However, knowledge represented in the Patient Data has always been transferred from the Patient at some point.

Fundamental concepts that are included within the implementation are histories. They can keep track not only of single configurations, like the state of an Asbru plan or the status of the patient, but also of traces of these configurations. Histories store those configurations together with the time, at which they occurred.

The control flow is specified in the Asbru and Abstracted Asbru specifications. They implement the state-chart written down in the Asbru semantics definition paper [7]. This kind of implementation, where Asbru plans are interpreted by

an interpretation engine and are separated from the configuration that is stored within the Asbru State is a very direct implementation of the Asbru semantics. Experience shows, that the separation of control flow and data increases the readability of the proof obligations, thus reducing the effort to learn Asbru verification.

Although not strictly within the scope of this publication, we want to provide an overview about the timely changes of the data structures. Access to these is in general limited to an access in a controlled way. This is depicted in Figure 2.

The calculus employed to verify the sequents is the ITL calculus described in [2]. This calculus divides a temporal step into two half steps, where one is made by the specified system, the other is made by the environment. The system step is made of all the changes, Asbru plans make, as is described within the Asbru semantics state-chart overview. That is, the system step consists of plan state changes, the spawning of new sub-plans and so on. During this half step, histories and Patient are not changed. This is, because the state change of the patient is not the responsibility of Asbru plans.

The only data structures that might change within the system step are the Asbru-State and the internal plan signals. Within the Asbru state data structure, every plan is only allowed to change its own state, as for the internal plan signals, plans are only allowed to change signals sent to their sub-plans. As this is guaranteed by the Asbru execution engine, it can be concluded, that no concurrent write access to two different Asbru state fields or signal fields will occur. This however has to be made explicit for formal verification. As Asbru plans are executed synchronous parallel, there is concurrent access to several data structures. A predicate sync has been defined, which specifies, which concurrent access is ordered and which results in a clash. See the specification ostore-sync for details.

More complex than the system half step is the environmental half step. The environment is responsible to take the outcome of the system step and log it into the histories. The environment is also responsible for advancing the time and to distinguish between micro and macro-steps. With the time advancing, setting the older history entries back in time is being done automatically. The remaining task is to set the current Asbru State to the current entry field within the Asbru State History. For properties without time annotations it has been established to be useful, to automatically discard the older Asbru State History fields. This reduces the amount of formulas on the sequent, therefore increasing the efficiency of TL calculations and improving readability.

Data transfer is different for the Patient Data History. Although data may be transfered from Patient, it is not necessarily done. A data transfer from Patient to the Patient Data History is seen as a measurement. Those measurements in general have to be initiated by Asbru plans, which is done by the start of a plan with plan type ask. Without a running ask plan, it is assumed that no measurements are taken and therefore the current entry in the PDH remains unchanged.

Within Patient all values that are stored may change independently of one another. This can be limited to some extend by so called effects of Asbru plans or by the modelling of background knowledge. Limitation may include a

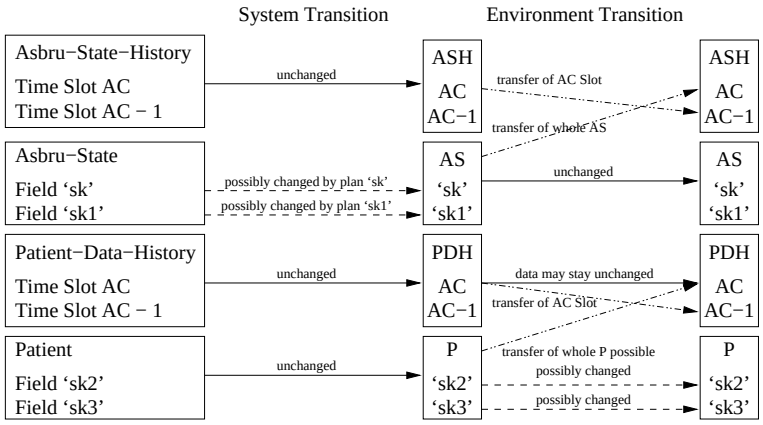


Figure 2: Dataflow in Asbru

complete prohibition of changes, quantitative changes (e.g. a drop between 5 and 15 mg per litre) or qualitative changes (value is raising).

The Asbru state has to remain unchanged during the environment transition with respect to all the plans that are currently executed. For some proof techniques, it is possible to avoid the direct execution of a subset of the Asbru plan hierarchy and the Asbru state may change for all those plans, which are part of the current hierarchy, yet not executed directly. This proof technique is called proof decomposition and is described in the verification tutorial “Asbru in KIV” [6].

8.2 Asbru state

8.2.1 plan-state-basic

The plan states as known in Asbru are defined in the specification plan-state-basic. They are defined as a data type with the values known from the Asbru plan state model.

8.2.2 plan-state

plan state is an enrichment to plan-state-basic, its intention to define derived predicates and functions here. Currently in this specification there are predicates defined to sum up certain semantical similar Asbru states. For example, there is a state-group terminated, which is a sum up of the states completed, rejected and aborted. Simplifier rules to deal with this state-group are defined here.

8.2.3 asbru-state-history-basic

This specification defines the mapping of the general histories, such that the dynamic function contained in each history entry maps from strings to Asbru states.

8.2.4 asbru-state-history

This specification houses simplification rules for dealing with Asbru state histories. One example is the dealing with intervals of time, during which the Asbru state remains unchanged. Another example are weakening rules, defining, under which circumstances entries in the Asbru state may be weakened to establish an invariant for application of induction.

8.3 Patient and patient data

8.3.1 data-value

data-value is a basic specification that defines the parameter for the patient, variable and the patient-data. This parameter is refined in the case study dependent files with the actual data types used in the case study. This way it is possible to keep the Asbru specification generic in the sense that it can be used as a library for all possible case studies.

Technically it has to be replaced by a sum type of all data types that occur in the case study.

8.3.2 patient-data-history-basic

This specification is an actualisation of the history with gdata-value as parameter sort. As the gdata-value itself has a parameter, the generated patient-data-history sort is still parametrised.

8.3.3 patient-data-history

The patient-data-history specification is an enrichment of the patient-data-history-basic specification. It has been designed as a collection of further simplifier rules. Currently it is an empty specification.

There is a major semantical difference between the patient and the patient data (sometimes also called patient record). While the patient data describes the knowledge of the medical staff about the patient, the patient itself contains the real data. There is a linkage between both, mainly that measurements in general transfer data from the patient to the patient records. It has been decided that the patient should not have a history.

Asbru decisions, that is plan state changes that rely on medical data, can only be made dependent on the patient data, not the patient. Indicators and intentions may also be evaluated over the patient itself.

The data transfer from the patient to the patient data has been somewhat controversial between the Asbru inventors and the formal methods experts. While the former insist, there is no possibility to force the doctor to take a measurement or forbid him to do so, the latter are not happy with completely indeterministic updates. The main reason for this is, that some properties rely on the number of measurements taken. For example one intention within the jaundice guideline is, that a certain blood value is only measured once within 24 hours. With indeterministic choice, this cannot be guaranteed. There is also a technical reason, that the indeterministic measurements lead to many case distinctions, two to the power of n, where n is the number of different values.

A consensus has been reached, that it is neither wanted to castrate the capabilities of Asbru here nor is it in the interest of the working group that properties cannot be proven. It has therefore been decided to make it case study dependent how data is transferred from the patient to the patient records. This will be added as an environment assumption to the individual sequent. An assumption, that should be linked to the Asbru ask plan. This assumption can be, that every ask has to be followed, no data may be transferred outside the ask plans or anything in between.

This provides the maximum degree of freedom and choice at the latest possible time. With some additional thought put into the formalisation, the indeterminism is mostly hidden from the sequent and will only lead to case distinctions if explicitly requested within the proof.

8.4 Variables

8.5 variables-basic

variables-basic is a refinement of the history with data-value as parameter sort. It closely resembles the patient-data-history-basic specification.

8.6 variables

The variables specification is an enrichment of the variables-basic specification. It has been designed as a collection of further simplifier rules. Currently it is an empty specification.

9 Time annotations

9.1 time-annotation-basic

Time annotations are an important Asbru feature to reason about time. A time annotation allows to formulate properties that access histories in a controlled way. Employing time annotations makes it possible to guarantee certain sanity conditions about time-reasoning. As stated in the Asbru semantics definition, time annotations introduce a somewhat three valued logic, in the sense, that they can be satisfied (true), not satisfiable (false) or satisfiable but not sat-

isfied (unknown). The time annotation basic specification houses the syntax definition of time annotations as a tuple of six (possibly infinite) integers, representing starting and finishing points and duration. Additionally, an asbru clock is contained, which is the reference point. Look at Section 9.3 to learn about complex reference points and their linkage to the evaluated and abstract time annotations.

9.2 time-annotation

The time annotation specification adds some basic reasoning about the sanity of time annotations. It adds the predicates legal and normal, which determine satisfiability on a very basic level. A time annotation is legal, if there exists an interval which is member of the time annotation. This is similar to state, that with the proper condition, it is possible to find a trace such that the condition linked with the time annotation is satisfied. Details about this reasoning are published in the definition of the semantics of the Asbru time annotations [8]. Also, details about the predicate normal, which is sort of a minimality constraint for time annotations are defined in that paper. A legal time annotation may be normalised without losing any traces and every normal time annotation is also legal.

9.3 Complex reference points in KIV

For every specification regarding time, i.e. asbru-clock-basic, asbru-clock, time-annotation-basic, time-annotation there is specifications with prefix “abstract”. These specifications deal with the complex asbru reference points. Asbru allows not only to specify absolute time points as reference points within time annotation but also the complex reference points *now* and enter/leave.

When evaluating time annotations, the reference point *now* is set to the time of evaluation, so it is constantly changing during evaluations. Time annotations with the reference point *now* cannot be evaluated directly but only after the reference time has been replaced by the time of evaluation. In a similar way, asbru time points enter and leave are treated. These constructs require two additional parameters, a plan-name and an Asbru state history. The time of enter(plan, state, ash) is the last time, the plan plan entered the given Asbru state.

Beside some additional constants specifications with the abstract prefix only specify the mapping between their respective asbru clocks and non abstract level. They are therefore not described in detail below.

In many specifications there are lemmas defined, prefixed by an aux and postfixed by a number. In general those lemmas are necessary as simplifier rules. Those lemmas will not be explained in detail, as they usually do not represent any semantically interesting property.

9.4 abstract-asbru-clock-basic

This specification provides a sum type of either an Asbru clock or the complex reference points *now* and enter/leave. These allow to specify relative time points with which it is possible to reference plan activations and the like of Asbru plans.

9.5 abstract-asbru-clock

This specification provides additional simplification rules to deal with abstract Asbru clocks, especially concerned with the evaluation of the complex time points.

9.6 Abstract-Time-Annotation-Basic

This specification defines a new data type, containing a time annotation, consisting of six integers (or infinity) in combination with one Asbru clock.

9.7 Abstract-Time-Annotation

Within this specification simplification rules are defined to deal with abstract time annotations. Especially cases, in which the abstract time annotation can be automatically transformed into an non abstract time annotation are dealt with here.

10 Conditions

10.1 condition-basic

Basis of the behaviour of all Asbru plans is a set of Asbru conditions. All critical plan state transitions not controlled by the super-plan plan are guarded by Asbru conditions. A condition consists of a higher order predicate, mapping the known state of a patient, the state of the Asbru plans and the values of the variables to a truth value. This truth value may be combined with a time annotation such that the combination checks, whether there is an interval in the bounds of the time-annotation and the predicate evaluates to true during the complete interval including its boundary points. If such an interval exists, the condition is satisfied. If such an interval does not exist, however the trace could be completed such that there is a point where the condition is satisfied, the condition is satisfiable. It has been decided to extend the definition of satisfiable such that a satisfied condition is still satisfiable.

Conditions can be combined with logical junctors and and or. Given a certain trace, an and combination of two conditions is satisfied, if both conditions are satisfied, it is satisfiable, if both underlying conditions are satisfiable. Given a certain trace, an or condition is satisfied, if one of both parts is satisfied and the combination is satisfiable, if one of the conditions is satisfiable.

In general a condition that is not satisfiable at a time t will not be satisfiable at any t' with t' later than t. There is only one exception to that rule. Once a plan is restarted all the complex reference points of this plan and all sub-plans are reset. With the reset of the reference point a condition can again become satisfiable. This behaviour is consistent with the idea behind the satisfiability of time outs and conditions. A condition should be satisfiable, if there is one path in the future, in which no timeout in the time annotation combined with the condition occurs. If it can be established, that such a timeout will occur in the future, regardless the changes in the condition from now on, it seems reasonable, that the same examination with the same time annotation somewhere in the future comes to the same conclusion. However, with a plan restart, the complex reference point may be evaluated to different time. It is also reasonable, that changes in the time annotation may lead to changes in the time-out behaviour. Additionally to the specification here, the Asbru language definition describes the logical junctor not and the derived operator xor. It has been tried to give a formal definition of the not junctor, however to the current point, no definition has been found that matches the requirements of the Asbru language designers and common sense. It has been decided unanimously with the Asbru developers to leave the definition of negated conditions out of the Asbru semantics definition for now, and come back to the issue, once it has been established, there is actually a need for it in a case study.

10.2 condition

This specification houses the semantics definitions described above. Additionally some predicates and functions have been specified that proved useful while verifying properties regarding conditions. Also some semantics checks have been done within this specification. It has been tried to establish the property, that for every condition, that is satisfiable, there is some time point in the future, such that it is possible for the condition at this point to be satisfied. Also, the property states, that the other direction also holds, that is, if a condition is satisfied at t, it was satisfiable at every t' with $t' < t$, given the reference point did not change from t' to t.

This proof has been finished at a very early stage of the project. In the mean time, parts of the underlying specifications were changed such that the proof has become invalid. There is strong confidence, that the changes did not affect the validity of the property, however, at the current stage of the project there is not enough time to verify the property again. Instead, it has been decided to postpone the re verification.

Further semantic tests have been done. One example was a test, whether the evaluation of a condition to a time t is independent of the choice of future traces. This is a reasonable property and verifying it, a flaw within the definition of the satisfied predicate was found. Under certain conditions evaluation of the predicate satisfiable was dependent on a future state.

The syntax of conditions is difficult to get used to at first. Therefore, we present an detailed example for a condition.

mk-acond(λ pdh, vh, ash, as, ac. pdh[ac] .bilirubin > phototherapy-recommendation)

This expression is a higher order predicate, as denoted by the λ expression. The variable designations that follow the λ are the local variables of the lemma. In this case, they are a patient data history, a variable history, an asbru state history, an asbru state and an asbru clock. In future versions of the implementation this signature might be changed.

Following the signature, concluded by a dot, the predicate is written down. In this case it is stated that from the patient data the field containing the bilirubin value should be selected. This value is a symbolic value of the concentration of the bilirubin in the blood and has to be greater than the symbolic level phototherapy-recommendation for this condition to be satisfied.

10.3 asbru-condition

Conditions in the sense of the Asbru semantics allow the user to override them or to prevent their evaluation to true. Each condition can be overridable, which is, the environment can force the system to treat a condition as if it was evaluated to true although it is not. There is a dual concept to allow the environment to forbid the system to proceed although a condition would have been evaluated to true. For these concepts to be implemented, an Asbru condition combines a condition with two booleans.

10.4 conditional

A conditional is a simplified version of a condition. It shares the principle of the condition, being a higher order logic predicate, mapping patient-data-history, variable-history, asbru-state-history, asbru-state and asbru-clock to a boolean. It lacks the possibility to combine these values with a time annotation or to merge several of these predicates together with and or or.

11 Plan body types

11.1 plan-type-basic

Within this specification a data type is defined, which holds the possible types of Asbru plan controls. These plan types are mainly those defined by the Asbru syntax definition. In addition to the standard plan types, one type on-abort-on-suspend has been defined for the convenience of the users. The semantical definition of these plan types can be found in the Asbru semantics definition.

11.2 plan-type

The patient specification is an enrichment of the plan-type-basic specification. It has been designed as a collection of further simplifier rules. Currently it is an empty specification.

12 Mandatory and optional sub-plans

12.1 wait-for-basic

This specification houses the syntactical specification of the Asbru wait for construct. The details about the semantics definition can be found in the wait-for specification.

12.2 wait-for

wait for constructs in Asbru allow to specify sub-plans, that are in some way critical for the execution of a plan. Usually a plan has to wait for some of its sub-plans before it is allowed to complete itself. A wait-for for a single plan is satisfied, once this plan is completed. It is no longer satisfiable, once the plan aborts. Logical and / or connections of wait-fors are self explanatory. Some explanation is needed for the negation of a wait for and therefore for the derived xor of two wait fors. There is the possibility, that the completion of a plan can be seen as a signal something bad has happened. In such a case the super-plan of such a plan specifies such a plan as wait-for-not. A wait-for-not is satisfied, as long as the plan has not yet completed and is satisfiable iff it is satisfied.

13 Plan synchronisation

13.1 plan-com-entry

A plan com entry is a technical specification, aggregating three boolean values into one data structure.

13.2 plan-com-basic

Within this specification, a list data type is actualised, such that it actualises a dynamic function to be of type string -> plan-com-entry. This specification is only technically relevant.

13.3 plan-com

Plan com gathers the three signals that might be sent from a super-plan to its respective sub-plan. These three signals are required for plans to proceed at certain, critical points. Some control types for asbru plans require the sub-plans to be synchronised, for example the parallel control which requires all sub-plans to switch from ready state to the activated state at once. This synchronisation is done via signals sent from the super-plan to the sub-plans.

Three synchronisation points are defined, one with the transition from inactive to considered, one with the transition from ready to activated and the last one

from the abort state back to consider. Those three signals are gathered in a data structure defined in the plan-com specification.

In recent implementations in KIV it has been decided to implement these signals as internal variables, shielding them from environmental influence. This has improved the efficiency as well as simplifying the sequents, as environmental non-interference does not have to be specified separately.

14 Environment interaction

14.1 bool-tupel

A bool tupel combines six boolean values into one data structure as a technical prerequisite to the environment aggregation.

14.2 environment-aggregation-part-basic

An environment aggregation part basic pairs two bool tuples such that the resulting data structure holds 12 signals. This is a technical prerequisite to the environment aggregation.

14.3 environment-aggregation-basic

The environment aggregation basic specification actualises a dynamic function, such that it is of type string -> environment aggregation part basic, therefore defining a data type necessary to deal with user interactions to Asbru conditions.

14.4 environment-aggregation

A plan is in some way controlled by other plans and the environment of the complete Asbru system. From the perspective of the individual plan, it is not important whether a signal originates from a concurrently running plan or the environment (i.e. medical staff). Therefore, for a plan all these signals are seen as environment signals. For technical reasons signals sent by the medical staff and signals sent by other plans have been separated with the environment aggregation containing all the environmental signals and the plam com data structure containing all the inter-plan communication.

Every condition in Asbru has the possibility to define it to be overridable or to require confirmation. The former states, that medical staff can decide to bypass this condition, even if it is not satisfied, while the latter requires a doctor explicitly consent with proceeding further in the execution. The first may be used if there is a weak preference not to proceed if a certain condition is not satisfied, but the doctor may decide, that at a given special case there is a reason to proceed. The latter may be used to describe a condition, that is valid in most cases but in some special cases, that have to be determined by the medical staff, it is not.

To be able to distinguish between signals for every plan, it has been decided to implement the overall data structure in the same way as the asbru state, refining a dynamic function to contain the 12 signals. The key to select the data fields out of the function is the plan-name.

15 Asbru plans

15.1 Syntax

15.1.1 asbru-def-basic

This specification houses the data structure representing Asbru plans. It has been decided to completely separate the plan execution from the definitions of Asbru plans. The semantics of the plan execution is therefore not implemented here, but in the specification Asbru. The plan execution semantics is designed along the definition of state charts.

A plan as designed here consists of the six Asbru conditions that guard the plan execution transitions in the Asbru semantics. Additionally it is determined of which type the Asbru plan is, whether or not it should be retried, once it is aborted, a list of its sub-plans, the wait for construct and a flag optional wait for.

The latter is to determine the plans behaviour regarding its already started sub-plans. Once all mandatory sub-plans of a plan are completed and its complete condition is satisfied, the plan may go to the completed state. However, it might be possible, the plan has also started non mandatory sub plans that are still running. If the plan completes with optional sub-plans running, this will result in an abortion of the still running sub-plans. If this is undesired, the plan can wait for the optional sub-plans, leaving none of them in aborted state.

15.1.2 asbru-def

The main concern of this specification is to define abbreviations for the satisfiability. With the consideration of the environment signals with the evaluation of the satisfiability of conditions, formulas defining this satisfiability have become confusingly complex. For the human verifier to have a better overview, it has been decided to define abbreviations filter-satisfied, filter-satisfiable, setup-satisfied and so on. Also, within this specification a function has been defined to map string-keys to asbru definitions. This function will later be partially specified for those keys that represent valid asbru definitions in a case study.

15.2 Semantics (plan state model)

15.2.1 Asbru-basic

This specification is a union of all specifications, necessary to implement the asbru semantics state-charts.

15.2.2 asbru

The asbru specification houses the implementation of the asbru semantics state-charts. The implementation can be seen as an asbru interpreter. It has been decided to keep the programs as close to state-charts as possible. This has been realized by implementing each state of the semantics state-chart with one procedure and these procedures may call other procedures along the transitions which are mentioned in the semantics state-chart. All of these procedures are implemented as one-step terminating.

For the user the real execution is therefore invisible. At any given state, all of the currently active plans will have a corresponding program called inactive#, considered#, possible# and so on running. After one step of the system, some assignments will have taken place. The user will then spot one program, corresponding to the new status of the corresponding asbru plan. Changes to that scheme are termination and spawning of asbru-plans. Changes to the plan states may also be seen in the predicate logic part of the sequent, that is in the asbru-state and asbru-state-history data structures.

16 Abstraction

16.1 asbru-abstracted

For some properties the assignments done by the asbru programs are too fine coarsed. It may be unimportant, how many steps a plan has spent in the evaluation phase and therefore an abstraction is possible, not distinguishing between possible, ready and considered state. These abstractions are highly experimental and should be used with caution at this time.

17 Auxiliary specifications

The specification hierarchy include some specifications which can be seen as auxiliary specifications. They can be existent to enhance readability by structuring the simplifier rules that are necessary. Others are there for technical reasons and may simply contain definitions of data type. Those specifications are described here together with their purpose mainly for reasons of completeness. Their understanding is not required to understand the Asbru implementation.

17.1 jgelem, prepair

jgelem and prepair are an auxiliary specification, needed for technical reasons to actualise a pair with the same data type as elem and data (i.e. as a pair of the same sort)

17.2 gdata-value

This specification is only technically relevant.

18 Conclusion

This implementation of the Asbru semantics has been developed over the course of 2 1/2 years. Focus of the development was on the ease of use for the verifier as well as the execution speed of the symbolic Asbru plans. The implementation is considered a success, as with the current version of the implementation it is possible to tackle even large hierarchies of plans, consisting of dozens of plans with multiple hierarchical layers.

A setback within this formalisation is the complexity of the different data structures, that store histories of system and patient configurations. However, these complex data types are necessary for the support of time annotations, which are a central concept of Asbru itself. Therefore it is our conclusion that all attempts to implement the semantics of an Asbru subset including time annotations will have to deal with similar concepts.

Several different verification techniques have been devised. These techniques range from a "head on" symbolic execution on the one hand and abstraction and generalisation techniques on the other hand, making use of modularisation technology. With this variety of methods it was possible to successfully deal with the real world case study of the treatment of breast cancer as used by the Dutch CBO.

References

[1] G. B. A. Benveniste. The synchronous approach to reactive and real-time systems. In *Another Look at Real-time programming*, Proceeding of IEEE, 1991.

[2] M. Balser. *Verifying Concurrent System with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction*. PhD thesis, University of Augsburg, Augsburg, Germany, 2005.

[3] M. Balser, C. Duelli, and W. Reif. Formal semantics of Asbru – An Overview. In *Proceedings of IDPT 2002*. Society for Design and Process Science, 2002.

[4] KIV homepage. `http://www.informatik.uni-augsburg.de/swt/kiv`.

[5] S. Miksch and K. Hammermüller. Asbru, a plan-representing language modelling time-oriented, skeletal plans in sport. In A. Baca, editor, *2nd International Symposium Computer Science in Sport*, 1999.

[6] J. Schmitt, M. Balser, and W. Reif. Interactive verification of Asbru - a tutorial. Technical Report 2006-3, University of Augsburg, February 2006. URL: `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/`.

[7] J. Schmitt, M. Balser, and W. Reif. Implementation of the Asbru semantics. Technical report, University of Augsburg, to appear. URL: `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/`.

[8] J. Schmitt, W. Reif, A. Seyfang, and S. Miksch. Temporal dimension of medical guidelines: The semantics of Asbru time annotations. In *Proceedings of European Conference on Artificial Intelligence (ECAI'06)*, to appear.

[9] A. Seyfang, R. Kosara, and S. Miksch. Asbru's reference manual, Asbru version 7.2, document revision 1. Technical report, Vienna University of Technology, Institute of Software Technology, 2000.