UNIVERSITÄT AUGSBURG
Universitätsbibliothek

OPEN ACCESS
OPUS AUGSBURG

Asbru in KIV v2.1 –A Tutorial

Jonathan Schmitt, Michael Balser, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

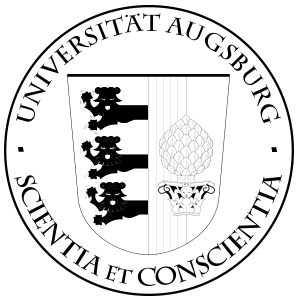
Schmitt, Jonathan, Michael Balser, and Wolfgang Reif. 2006. “Asbru in KIV v2.1 –A Tutorial.” Augsburg: Universität Augsburg.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/

Universität Augsburg



Asbru in KIV v2.1 – A Tutorial

J. Schmitt, M. Balser, W. Reif

Report 2006-03 March 2006

Institut für Informatik
D-86135 Augsburg

Copyright © J. Schmitt, M. Balser, W. Reif
Institut für Informatik
Universität Augsburg
D–86135 Augsburg, Germany
http://www.Informatik.Uni-Augsburg.DE
— all rights reserved —

1 Introduction

Aim of the Protocure II[1] project is to improve the quality of medical guidelines. Currently, medical guidelines are informal texts, written by an expert group, based on available medical facts. It will come to no surprise, that anomalies may remain in guidelines. These can be ambiguities in the text, modelling errors or even wrong data. This is, where the Protocure II project offers tools and methods to further improve the guideline quality. Foundation for these improvements is a stepwise formalisation of the informal guideline text, such that knowledge engineering and finally theoretical computer science methods can be applied.

This report focusses on the last step in the chain of methods, namely the verification, that the formalised guideline adheres to correctness properties. For this, it is necessary, that the guideline is already available in a formalised language. Within Protocure II, we choose to model the guideline in the planing language Asbru [5].

Before being able to verify properties over systems in a given non-standard representation like Asbru, it is necessary to clearly define the semantics of such a system. Although this task might be quite difficult, the task following that, to include the defined semantics in a theorem prover or a model checker might be even more challenging.

After the language definition and semantics of Asbru Light [3] had been revised to match the needs of the Protocure II case study breast cancer, it was necessary to implement it in KIV.

As Asbru has been integrated in KIV using an already implemented formalism, namely parallel programs, the main difficulty was to find representations for data structures that match Asbru on the one hand and at the same time go along with general automation techniques already implemented in KIV. Result of this strategy to rely as much as possible on already existing and well tried parts of the KIV system was, that for the completion of this task virtually no implementation effort in the KIV base system was necessary. This was possible, because Asbru was specified in KIV instead of implemented.

For a verifier to be able to prove properties, it is also necessary to know about techniques to apply when proving a sequent to be correct. Therefore some time has been invested to elaborated proof techniques to cover all Asbru proof aspects, ranging from simple user performed plans to hierarchies with several hierarchical layers and dozens of contributing plans.

This document therefore intends to give the user a very short and pragmatic introduction into the most important aspects of the implemented Asbru semantic, afterwards describing the different techniques necessary to make use of the interactive theorem prover KIV to prove properties.

[1] www.protocure.org

This technical report describes the Asbru implementation, proofs and techniques current at the 03/06/2006. This implementation has been designated as version 2.1 which is not related to the version of the Asbru language.

2 Asbru semantics

The Asbru semantics are described in detail in [3]. The Implementation of the semantic in the KIV system is described in [4]. We will therefore only give a short introduction into the most important aspects of the Asbru semantic as implemented in KIV and refer the interested reader to the more detailed papers.

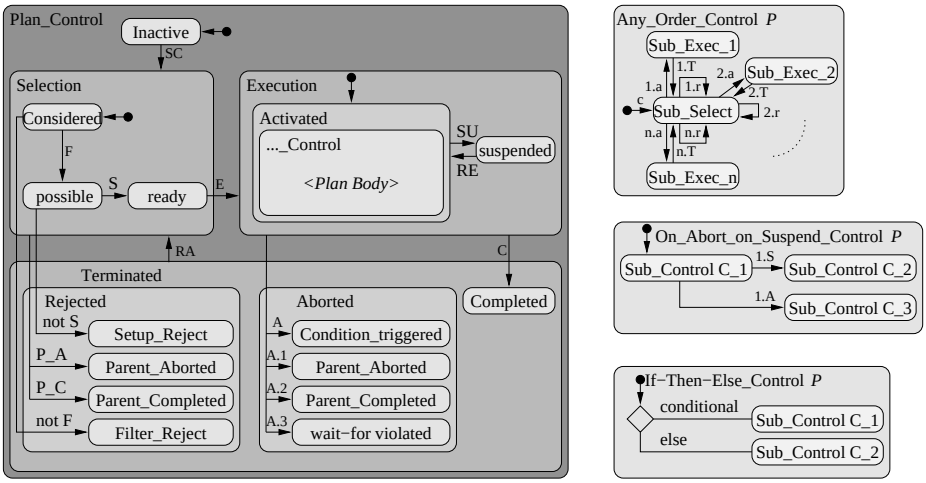


Fig. 1. State transitions of Asbru plans

Basis of the Asbru semantics is the concept of plans with a defined plan state model. Asbru plans may advance through the plan state model in a way, defined by the statechart depicted in Figure 1. All transitions within this statechart are guarded, where the guards `SC`, `RA` and `E` describe external signals, which are sent to the plan by its respective parent. Within this implementation the signals are encapsulated in a data structure defined in the specification `plan-com`.

All the other guards are related to the evaluation of Asbru conditions which define the behaviour of Asbru plans. Syntax and semantics of Asbru conditions are defined in specification `Abstract-Asbru-Condition`, where a higher order predicate is paired with two booleans, which model the flags `overridable` and `manual-activation` as defined by the asbru semantics.

Six of those condition define the overall behaviour of an Asbru plan, that is, the way it advances through the evaluation phase and the circumstances under which

it terminates or suspends. Additional behaviour of the plan is described by the plan type. This, along with a list of potential children fills in the blanks within the activation phase in the statechart in Figure 1. There are further parameters of an Asbru plan that describe the behaviour even more detailed. For example, a flag retry specifies how a plan should react, given one of its children aborts.

Behaviour of Asbru plans is defined within the specification `Asbru`, where the statechart of Figure 1 is implemented by parallel programs. This has been done in a generic way, such that all Asbru plans use the same set of parallel programs for their evaluation. It has been decided to hide the technical details from the verifier. This has been realised by designing the defining parallel programs in a way such that the programs are recursive and always terminating within one step. Therefore, a verifier will only be confronted with a very small set of different procedure names, not the program code itself. Every non-passive state in the statechart, which is, `inactive, considered, possible, ready, activated` and `suspended` is represented by a separate program, encapsulating the possibilities of the representing state. Passive states do not require further procedures.

Asbru procedures are designed similar to state-charts, where state and control are separated. The overall state in the case of Asbru can be divided in several parts, namely the state of all the currently running plans, which is defined in specification `asbru-state-history`, the currently known condition of the patient, defined in `patient-data-history`, technical information like variables, stored in a data structure defined in `variable-history`. Finally, communication between a plan and its relatives or the plan and the environment is part of the state. While the former is defined in a data structure defined in specification `plan-com`, the latter can be found in a data structure in specification `environment-aggregation`. The only information encoded in the programs is the way, a plan may react within the next step, so the plan "knows" its own state from the program constellation, but no more.

3 Proof obligations

A proof obligation or property may for example describe, that an Asbru plan adheres to an indicator, which is a medical quality assurance instrument. It may also be a technical property, for example, that a plan always terminates at some point or that it does not disturb execution of another plan running in parallel. Proof obligations consist of two parts; one of the parts of the property describes the behaviour of the Asbru plan hierarchy, the other one the expected behaviour. Behaviour of the Asbru plan hierarchy can be further broken down into system description, system configuration and environment assumption.

3.1 System description

Within our implementation of the Asbru semantic, the system description consists of synchronous parallel running programs. Each program represents one Asbru plan, therefore plan hierarchies are represented by a number of parallel running plans.

As the execution of Asbru plans is somewhat generic, all plan types are represented by the same set of programs. Procedures vary in the parameters with which they are called and every different active state has its own program representation. There is one procedure describing the behaviour of an `inactive` plan, one describing the behaviour of a `considered` plan and so on. Details of the execution state are also integrated into the procedure parameters. These include, for example, the currently running children of the plan.

Although the plans work on different aspects of the global state regarding their own advance through the plan state hierarchy, it is not necessary to launch the plans with different Asbru state variables, as the state variables contain the current state of all Asbru plans. Each plan selects values relevant for itself. For one plan, this data is the state of the parent and all the children, which has influence on the abortion and completion behaviour of the plan. The Asbru plan procedures have been carefully designed such that concurrent write access to the same data fields cannot happen.

Within a proof obligation, the procedures look the following way:

considered#(sk, sk0; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG)

activated#(sk, sk0, skx1, skx2, skx3, n, m;
Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG, PC0)

The first part of the example represents a considered Asbru plan, while the second represents an active plan, a state which is further elaborated. The semicolon in the procedures separate the value parameter in the first half from the var parameters in the second half. The technical difference is, that operations on the value parameters are side effect free, so assignments to value parameters are lost after termination of the procedure and only have effects within the procedure and possibly sub-procedures. The variables `sk` and `sk0` are of type string and contain the name of the plan itself (`sk`) and the name of the parent plan (`sk0`).

The dynamic variables `Tick, Patient, PDH, VH, AC, ASH, AS, PC` and `EAGG` are equal for all Asbru plans on the sequent, that is, all Asbru plans work on the same variables here. Most of these variables are considered read-only, with the exception of the `AS` variable, representing the Asbru state, where each plan is required to enter its current state. The variables are mostly required for the plans to be able to evaluate their conditions.

Plans leaving the evaluation phase bound for the activation phase require additional parameters. The three lists `skx1, skx2, skx3`, that accompany an active

plan depict the subplans of this plan and describe to some extend the status, in which they are. The `skx1` list contains all subplans that are currently active themselves, the `skx2` list those, that are in the evaluation phase and the `skx3` list contains all subplans, yet in inactive state. Plans that are terminated do not appear in those lists. The natural number counters, `n` and `m`, are necessary when dealing with cyclical plans. For all other plan types, they are ignored. This technical report does omit details about cyclical plans and will focus on the other plan types.

As can be seen in the example, the `activated` procedure has an additional `PC0` variable. Figure 2 visualises the view of parent and child towards these variables. As can be seen there, the `PC` variable, which is present in all asbru procedures is merely for receiving signals from a (possibly anonymous) parent, while the second variable, `PC0`, is used to send signals to the children.

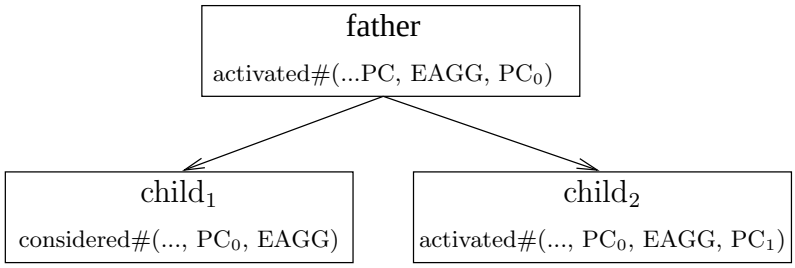


Fig. 2. internalised PC variables

3.2 System configuration

As described in Sect. 2, Asbru procedures are defined in a way similar to statecharts. Therefore, the current state of the system can not be read from program pointers. Instead, the state is stored in predicate logic formulas. These formulas are called system configuration. It has been decided to combine similar parts of the system configuration into one variable. Five different types of data have been identified, that have to be considered.

Asbru state First of these types is the state of the Asbru plans, currently running. This has been called the Asbru state. The history of Asbru states is necessary when dealing with time annotations. The representing data structure is called Asbru state history. In principle, an Asbru state consists of a mapping of plannames and plan status. A mapping may look this way:

AS[‘praoa’] = considered

AS[‘toan’] = activated

There are two ways how to write down the current system configuration, an explicit one and an implicit one. First examples of verification had been done with the explicit denotation, however this concept has been found to be inadequate when combined with the proof decomposition described later on. Therefore with all currently conducted verification examples, the implicit denotation has been chosen.

explicit
AS = as0[‘praoa’, considered]
[‘toan’, activated]
vs. implicit
AS[‘praoa’] = considered
AS[‘toan’] = activated

As can be seen above, the explicit notion of the Asbru state is done by taking an basic Asbru state `as0` for all the subplans that are not explicitly mentioned. This concept of ‘invisible’ plans is especially important when dealing with proof decomposition. The basic Asbru state is then updated.

Asbru state history and Asbru state have been separated because of the concurrent write access of several plans. This issue can be handled better in less complex data structures as the Asbru state versus the Asbru state history, where the latter is a generated data type with pairs of Asbru state and time variables.

Variable, patient and patient data Patient data is the knowledge that has been gathered from the patient. The structure of the denotation of the data is similar to the one used with the Asbru state history. However, as there is no concurrent write access to the patient data, the separation between the current state and history states is not necessary.

It is important to realize, that the patient data does not necessarily represent the patients real condition! Instead, at certain time points measurements to the patient may be made and therefore the data transferred from patient to patient data. From the authors point of view this was the best possible mapping of the real world to this model. As with reality, the patient, not the patient data is affected by treatments and the result of treatment are not automatically visible to the medical personal.

As with the Asbru state, there is an explicit and implicit way of writing it down, with only the implicit being used. The example depicts, that at current state the measurement called bilirubin is at level observation.

```
(PDH[AC][‘bilirubin’]) .val = observation
```

Variables are conceptual and technical similar to patient data, however they do not represent physical measured data, instead they represent the result of calculations, return values of plans and so on.

Environment aggregation and plan communication The environment aggregation summarises all signals that may be sent to the plans by the environment, which is the medical personal. Those signals do not include the signals sent by a father to its children (for example regarding plan activation). Signals stored in the environment aggregation are the signals to determine the evaluation of manual activated or overridable asbru conditions.

Each entry in the environment aggregation is linked to a plan via its key and consists of two sets of six signals each, one `confirmation` required and one `overridable` signal for each condition. The following example depicts the knowledge, that a plan called pob is currently receiving its override signal to its filter condition:

EAGG[‘pob’] .override .filter

Signals, sent from a parent to its child are stored within the `Plan Com` data structure. Should the plan pob receive its consider signal from its parent, it could be written on the sequent as

PC[‘pob’] .consider

but may be rewritten to

PC[‘pob’] = mk-pce(true, boolvar, boolvar0)

`mk-pce` is the constructor of an plan-com entry and takes three signals, `consider`, `activate` and `retry`, which correspond to the `SC, E` and `RA` signals respectively in Figure 1.

3.3 Environment assumption

In principle the environment is allowed to change every variable arbitrarily. Usually this has to be forbidden to some extend. The environment assumption is therefore a temporal logic formula restricting the behaviour of the environment. Usually this restriction comes in three different flavours, first to forbid the environment completely to change a variable, second to force the environment to change a variable in a deterministic way, for example, advancing a clock, and third to guarantee certain variable assignments in a nondeterministic way, for example guarantee the existence of a time, where a signal is sent.

One example for the first case is the Asbru state. This data structure is only to be accessed by the plan representing procedures. The environment is not allowed

to change Asbru state fields that are associated with plans for which procedures are running. Such an environment assumption can be written down as follows:

$$\Box\ (\quad \text{AS''}[\text{`praoa'}] = \text{AS'}[\text{`praoa'}] \\ \quad \wedge\ \text{AS''}[\text{`toan'}] \ \ = \text{AS'}[\text{`toan'}])$$

Another example would be, that the father of a running plan, that is not written on the sequent itself, is never suspended. This still allows the father to terminate or run through. Such a property is written down like

$$\Box\ \text{AS''}[\text{`dwa'}] \neq \text{suspended}$$

An example for the second case, changing a variable in a deterministic way, is the logging of the Asbru state history. As the plans only change the Asbru state, not the history. This logging is done by the environment.

$$\Box\ (\quad \text{ASH''}[\text{AC''}][\text{`pob'}] = \text{AS'}[\text{`pob'}] \\ \quad \wedge\ \text{ASH''}[\text{AC''}][\text{`ob'}] \ \ = \text{AS'}[\text{`ob'}])$$

The third example are liveness properties, stating for example that user performed plans are finally ended by the medical personal. Such properties are typically leads to properties.

$$\Box\ (\text{ASH}[\text{AC}][\text{`ob'}] = \text{ready} \rightarrow \Diamond\ \text{PC}[\text{`ob'}]\ .\text{consider})$$

4 Symbolic Execution of Asbru

Our proof method is based on a sequent calculus with calculus rules of the following form:

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \ldots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}\ \mathit{name}\ .$$

Rules are applied bottom-up. Rule *name* refines a given conclusion $\Gamma \vdash \Delta$ with n premises $\Gamma_i \vdash \Delta_i$. Furthermore, we heavily rely on the possibility to rewrite sub-formulas. A rewrite rule is given as

$$\mathit{name:}\ \varphi \ \leftrightarrow\ \psi\ .$$

With this rule, formula φ can be replaced by an equivalent formula ψ anywhere within a given sequent.

4.1 Normal form

The idea of symbolic execution of arbitrary temporal formulas – Asbru plans being just a special case thereof – is to normalise all the temporal formulas of a given sequent by rewriting the formulas to a so called normal form which separates the possible first transitions and the corresponding temporal formulas describing the system in the next state. Afterwards, an overall step for the whole sequent is executed (see below). More formally, an Asbru plan (or temporal formula) is rewritten to a formula of the following type

$$\tau \wedge \circ \varphi$$

with τ being a formula in predicate logic describing a transition as a relation between unprimed, primed and double primed variables. In general, a system may also terminate, i.e., under certain conditions, the current state may be the last. Furthermore, the next transition can be nondeterministic, i.e., different τ_i with corresponding φ_i may exist describing the possible transitions and corresponding next steps. Finally, there may exist a link between the transition τ_i and system φ_i which cannot be expressed as a relation between unprimed, primed, and double primed variables in the transition alone. This link is captured in existentially quantified static variables $\boldsymbol{X}_i$ which occur in both τ_i and φ_i. The general pattern to separate the first transitions of a given temporal formula is

$$\tau_0 \wedge \textbf{last} \vee \bigvee_{i=1}^{n} (\exists\, \boldsymbol{X}_i.\ \tau_i \wedge \circ\, \varphi_i)\ .$$

We will refer to this general pattern as normal form.

4.2 Rewriting temporal operators to normal form

We have defined a set of rules to rewrite arbitrary temporal formulas to normal form [2]. To automatically rewrite the formula, select an arbitrary sub-formula and apply rule *normal form* as shown in Figure 3.

An alternative to rewriting the formula to normal form in a single step is to apply rules to the top level operator only. For every temporal operator there is a rule to execute the operator, e.g. rule *if* is applicable for a conditional **if** φ **then** ψ_1 **else** ψ_2, rule *if*, for a procedure rule *call* can be applied (see Figure 4).

In both cases, it is important to simplify the result afterwards. Therefore, it is recommended to use the predefined set of heuristics *TL Heuristics* to automatically apply all of the rules simplifying the result.

4.3 Executing an overall step

Assume that the antecedent of a sequent has been rewritten to normal form. Further assume – to keep it simple – that the succedent is empty. (This can be

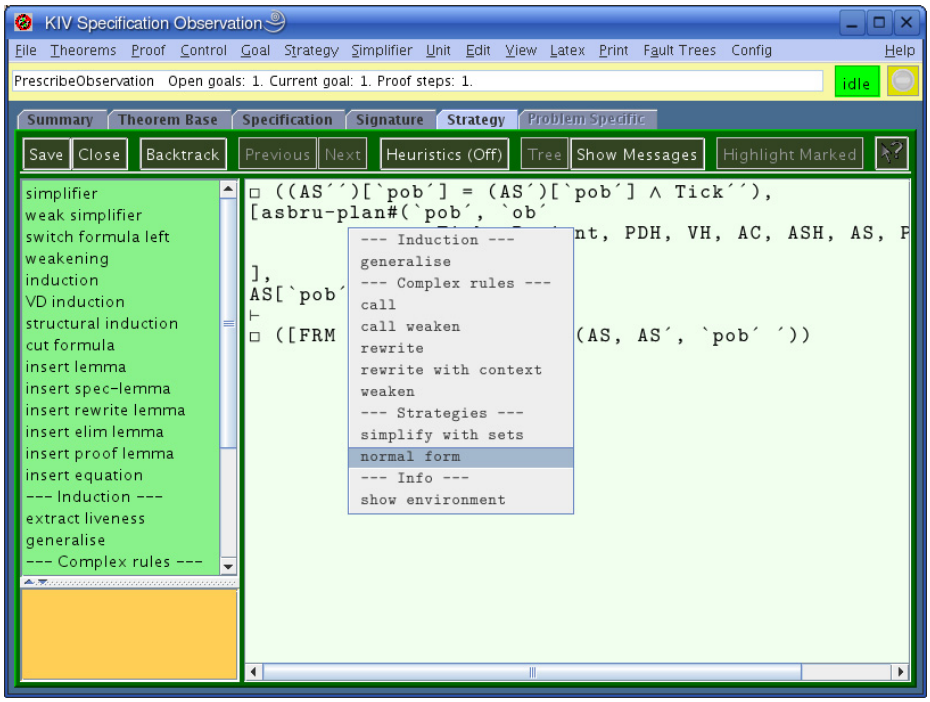


Fig. 3. Rewriting sub-formulas to normal form

$$\frac{\varphi, \Gamma \vdash \Delta \quad \psi, \Gamma \vdash \Delta}{\varphi \vee \psi, \Gamma \vdash \Delta}\ \mathit{dis}\ l \qquad \frac{\varphi_X^{X_0}, \Gamma \vdash \Delta}{\exists\, X.\ \varphi, \Gamma \vdash \Delta}\ \mathit{ex}\ l$$

where X_0 fresh with respect to $\mathrm{free}(\varphi) \setminus \{X\} \cup \mathrm{free}(\Gamma, \Delta)$

$$\frac{\tau_{A,A',A''}^{X,X,X} \vdash}{\tau, \mathbf{last} \vdash}\ \mathit{lst} \qquad \frac{\tau_{A,A',A''}^{X_1,X_2,A}, \varphi}{\tau, \circ\, \varphi \vdash}\ \mathit{stp}$$

where X, X_1, X_2 fresh with respect to $\mathrm{free}(\rho, \varphi)$

Table 2. Rules for executing an overall step

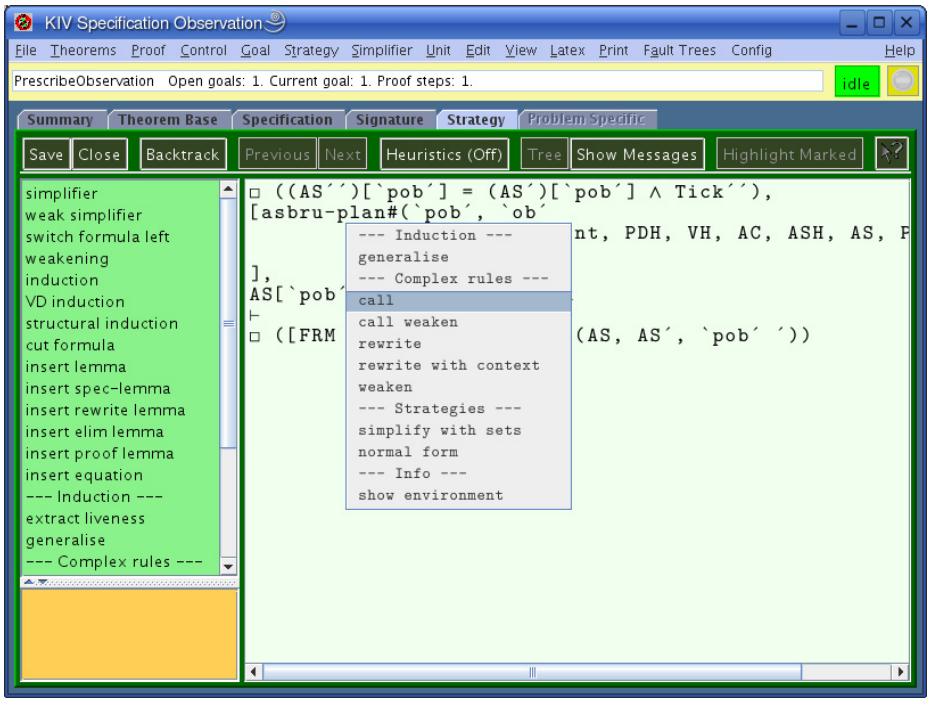


Fig. 4. Rewriting the top-level operator

assumed as formulas in the succedent are equivalent to negated formulas in the antecedent. Furthermore, several formulas in the antecedent can be combined to a single normal form.)

$$\tau_0 \wedge \mathbf{last} \vee \bigvee_{i=1}^{n} (\exists\, \boldsymbol{X}_i.\ \tau_i \wedge \circ\, \varphi_i) \vdash$$

With the two rules *dis l* and *ex l* of Table 2, disjunction and quantification can be eliminated. For the remaining premises,

$$\tau_0 \wedge \mathbf{last} \vdash \qquad \tau_i \wedge \circ\, \varphi_i \vdash$$

the two rules *lst* and *stp* can be applied. If execution terminates, all free dynamic variables A – no matter, if they are unprimed, primed or double primed – are replaced by fresh static variables X. The result is a formula in pure predicate logic with static variables only, which can be proven with standard first order reasoning. Rule *stp* advances a step in the trace. The values of the dynamic variables A and A' in the old state are stored in fresh static variables X_1 and X_2. Double primed variables are unprimed variables in the next state. Finally, the leading next operators are discarded. The proof method now continues with the execution of φ_i.

We have implemented a rule *step* which automatically rewrites every temporal formula of the sequent to normal form, and executes an overall step. Furthermore, the predefined set of heuristics *TL Heuristics + Exec* includes a heuristic to automatically execute steps. The heuristic executes steps breadth first.

4.4 Executing Asbru

We have defined procedures to implement the semantics of Asbru (see Sect. 2). These procedures can be executed with the existing set of rewrite rules.

It is recommended to first try to execute a step with rule *step*. However, if normalising all of the temporal formulas of the given sequent results in too many case distinctions, the application of *step* takes too long. In this case, abort the application of *step* with the stop button and execute the procedure defining the Asbru plan with manual application of rules. It is recommended to use the set of heuristics *TL Heuristics* to ensure that those rewrite rules which do not give case distinctions are automatically applied. It remains to expand procedures and to normalise those conditionals for which the conditions cannot be decided automatically.

4.5 Simplifier rules

Simplifier rules for Asbru plans Axiom `-def` defines plan Observation

toan-def:

```
  asbru('toan')
= mk-asbru-def(
      mk-aasbruc(
          mk-acond(λ pdh, vh, ash, as, ac.
                       pdh[ac]['parameter-any-forms-of-metastasis-in-the-SN'] .val,
                       default-ata),
          false, false),
      setup_condition,
      suspend_condition,
      resume_condition,
      abort_condition,
      complete_condition,
      anyorder,
      false,
      'praoa' + 'and' ',
      wait-for-n(1,'praoa' + 'and' + []),
      false); used for: s, ls;
```

For verification it is inefficient to replace every reference to plan Observation with the complete definition. Instead, additional simplifier rules can be defined which rewrite the different slots of the definition.

```
toan-filter:    asbru('toan').filter
            = mk-aasbruc(
                  mk-acond(λ pdh, vh, ash, as, ac.
                               pdh[ac]['parameter-any-forms-of-metastasis-in-the-SN'] .val,
                               default-ata),
                  false, false),
toan-setup      : asbru('toan').setup       = setup_condition;
toan-suspend    : asbru('toan').suspend     = suspend_condition;
toan-reactivate : asbru('toan').reactivate  = resume_condition;
toan-abort      : asbru('toan').abort       = abort_condition;
toan-complete   : asbru('toan').complete    = complete_condition;
toan-type       : asbru('toan').type        = anyorder;
toan-retry      : asbru('toan').retry       = false;
toan-subplans   : asbru('toan').subplans    = 'praoa' + 'and' ';
toan-waitfor    : asbru('toan').waitfor     = wait-for-n(1, 'praoa' + 'and' ');
toan-opt-wf     : asbru('toan').opt-wf      = false;
```

This strategy of simplifier rules has an additional advantage. Aside from the faster execution and simplification it reduces the effort to reprove properties after plans have been changed. It is therefore recommended to define a set of such simplifier rules for every Asbru plan.

Simplifier rules for Asbru semantics As KIV generally applies rewrite rule, after a syntactical match between the rule precondition and some formulas on

the sequent has been achieved, it is important to try to define and achieve a canonical form of the sequent.

This allows to establish the validity of the premise and thus the automated closing of open goals in certain cases. Also, it is possible to identify non information carrying formulas and eliminate them from the sequent. Consider the following example

$\vdash$ ps = inactive $\rightarrow$ (ps = considered $\leftrightarrow$ false)

Precondition of this rule is, that a plan state variable is assigned the value 'inactive'. If this is written down on the sequent, then all occurrences of formulas, where the same plan state is set to considered are replaced by false. This can have two possible effects. One, if the rewritten formula is in the succedent of the sequent, the result will be an instance of a rule called `false right`, which eliminates the false, thus clearing up the sequent. Should, however, the rewritten formula exist in the antecedent, then an application of rule `false left` would be possible and the goal would be immediately closed.

Another way of simplification is to convert terms to a canonical form, for example when considering multiple updates. In practice it is not wanted to have a too large number of simplifier rules for several reasons, one of them being the reduced efficiency. Reducing terms to a canonical form allows to formulate only a limited number of simplifier rules for this very form instead of dozens of rules for each special case. One example of such reducing rules is written down below.

$\vdash$ sk < sk1 $\rightarrow$ (sk1 ' + sk ' = sk ' + sk1 ')
$\vdash$ sk ' + sk ' = sk '

This rule would result in the ordering of sets via something like the bubble sort algorithm. As the simplifier in KIV applies such simplification rules also to subformulas, it is not necessary to define additional rules where one has to consider the set to contain more than two elements.

The second rule in conjunction with the first rule eliminates all doubly inserted elements, such that all sets on the sequent will be written down in a canonical way. This may be necessary because the test for equality of sets can be ascribed to a test for syntactical equivalence.

Sometimes the precondition of a rule is not written down on the sequent syntactically but can be deducted from knowledge on the sequent. Take for example the above mentioned precondition `sk < sk1`. Although, given < to be the lexicographical alphabetical ordering, it is true that `'ob' < 'rt'`, this will in principle not be written down on the sequent. Therefore, KIV allows for the formulation of more general simplifier rules.

sk < sk1 $\vdash$ (sk1 ' + sk ' = sk ' + sk1 ')

Can be used as a simplifier rule, where for every instance of `sk1 ' + sk '` it is tried to prove that `sk` $<$ `sk1`. This, of course is much less efficient and in general such a proof will not be doable, as soon as variables are involved.

Therefore, there is a third possibility for such rules to be formulated, which is

sk < sk1 $\vdash$ ((* sk1) ' + (* sk) ' = sk ' + sk1 ')

where the asterisks meaning is, that sk has to be a constant, not a variable. As the ordering on concrete strings is always decidable, this will only result in a minor efficiency loss while the simplifier is working.

Many such simplifier rules for the basic data types are already defined in the base libraries and even more where defined within the Asbru part of the library to deal with the specifics of Asbru. In general it should be thoroughly evaluated, whether new simplifier rules are useful in principle or not. One has to be aware that by inserting a simplifier rule within the library, the behaviour of the simplifier might be changed for more verifiers than oneself. This could result in the replay of proofs to be less automated and even in cyclic application of simplifier rules, which results in an unusable simplifier.

Much effort has been spent to make the simplification strategy as simple and efficient as possible, so in principle it is strongly suggested to consult with the designers of the Asbru semantics before changing it.

4.6 Induction

The execution of Asbru plans often loops. For example, if a plan is activated, it normally waits for its complete or abort condition to be satisfied. Also, if a plan is in ready state, it waits for its parent to send the activate signal. Refer to the state chart describing the semantics of Asbru [3] for more possible cycles.

In case of a cycle, induction must be used to close the proof. Induction is initiated with rule *VD induction*. If a safety property is verified, this safety property can be used for induction. Otherwise, an expression of type `nat` must be provided which is ensured to decrease during execution. After the loop has been executed, induction can be applied by selection of the corresponding node in the proof tree where execution of the loop has started. Right click on the corresponding node and choose *apply VD induction* (see Figure 5).

4.7 Heuristics

Heuristics define a set of rules that are automatically applied to a sequent. For example, it is recommended to automatically apply simplification rules to a sequent after execution of a step is completed. With simplification rules, a number of sequents can be closed automatically, the remaining sequents are in

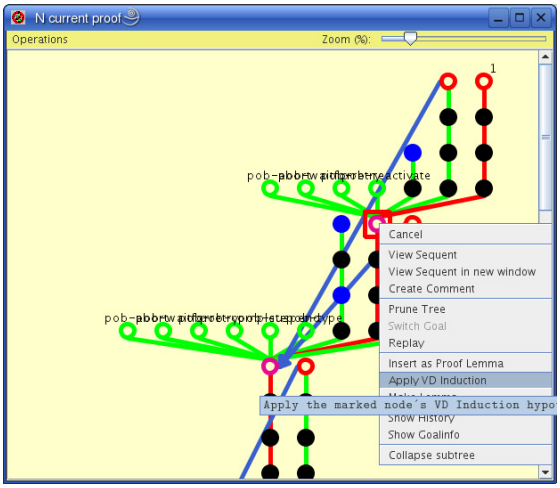


Fig. 5. Using *apply VD induction*

general more readable. Use the predefined heuristic set *TL Heuristics* for this purpose.

Sometimes rule applications can make proofs more difficult or more painstaking. Consider an example, where the real state of the system is hidden in a disjunction, such that $\texttt{state} \equiv \texttt{state}_1 \vee \texttt{state}_2$. In that case, it might be useful to split the disjunction, because the simplifier can close both resulting goals or both resulting goals have to be treated in a different way.

On the other hand, it might also be possible to close the goal without knowing details about the state, for example because there is a contradiction within another set of PL formulas.

Therefore, there is no general rule, when to split case distinctions and when not to do it. Such decisions can be automated by the choice of simplifier rules. One could for example especially select or deselect a set of rules containing a rule case distinction, or do so more specialised by stating, rule case distinction should only be applied, when the formula to process has a certain syntactical form.

For simple properties, it is possible to also automate step execution and the application of induction. Use the predefined heuristic set *TL Heuristics + Exec* for this purpose. However, the heuristics do not automatically generalise goals. Therefore, if it is necessary to generalise the current state, and also if it is necessary to manually simplify first order formulas, it is recommended to manually execute steps.

5 Verifying a single Asbru plan

We try to verify a simple property of the Asbru plan 'sentinel node procedure, patient information', abbreviated to `'SNpi'`. The property is named `SNpi-Intention`.

5.1 proof obligation

(: system configuration :)
AS['SNpi'] = inactive,
PC['SNpi'] .consider,

(: system description :)
[inactive#('SNpi', sk; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG)],

(: environment assumption :)
□ AS"['SNpi'] = AS'['SNpi']

⊢ (: property :)
□ ([Frame AS, Tick]
∧ frame(AS', AS, 'SNpi' '));

As can be seen in the proof obligation, the verification begins with the `'SNpi'` plan in the inactive state with its consider signal being sent by the anonymous parent *sk*. This can be read from the state of *PC*, where the consider signal for `'SNpi'` is set and *AS*, which is set to `inactive` in the slot concerned with `'SNpi'`. Also, the procedure describing the system is the `inactive#` procedure. In this example, the only assumption for the environment is, that it does not change the *AS* field of `'SNpi'`.

5.2 First step

The proof obligation as described above can be seen as node 1 in Fig. 6. We now apply a temporal logic step. According to the scheme, described in Sect. 4. Possible results of this step can be seen from Fig. 1. From the `inactive` state, the step started from, it is only possible to move forward to state `considered`, as the `SC` is satisfied. The resulting proof obligation is marked as node 2 in Figure 6

Technically, this is what happens during the system transition of the TL step: The `'SNpi'` field of the *AS* variable is set to `considered`, while the procedure representing the plan is changed to `considered#`. After the system transition follows the environment transition. There, all variables but *AS* are changed arbitrarily. For *AS*, the slots other than `'SNpi'` may again be changed arbitrarily.

After complete execution of the TL step, knowledge about the current status of the consider signal of `'SNpi'` is lost. Some static variables may remain on the sequent, describing some intermediate states. However, in the given goal, these static variables are of no importance and can be discarded manually to improve the readability of the sequent.

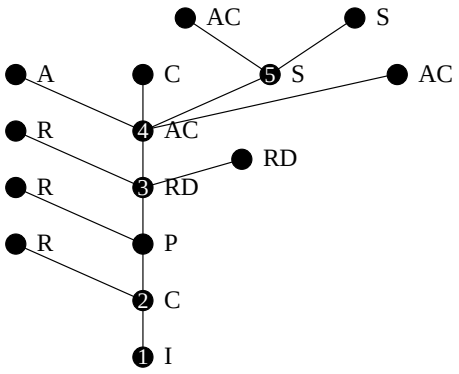


Fig. 6. Proof tree for SNpi-intention

An example for such static variables is given below, where some knowledge about intermediate states of the parent of `'SNpi'` is still written on the sequent.

[var PC0 = init
in activated#('SNpi', sk0, [], [], asbru(sk) .subplans, 1, 1
; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG, PC0],
INDHYP-nat(N + 1),
as[sk0] ≠ aborted, as[sk0] ≠ completed

5.3 Further steps

In every further step, the possibility of plan `'SNpi'` terminating has to be considered. Once plan `'SNpi'` terminates, its representing program will terminate. This will result in a first order proof obligation, which can be dealt with automatically by the simplifier.

Execution continues until plan `'SNpi'` is `ready`, to be seen in node 3 in Fig. 6. The proof obligation there is

(: system configuration :)
AS['SNpi'] = ready,
(: system description :)
[ready#('SNpi', sk; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG)],
(: environment assumption :)
□ AS"['SNpi'] = AS'['SNpi']
⊢ (: property :)
□ ([Frame AS, Tick]
∧ frame(AS', AS, 'SNpi' '));

After executing the next TL step, we receive two proof obligations. In the first, the activate signal has been sent and `'SNpi'`is `activated`:

activated#('SNpi', sk, [], [], [], 1;
Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG, PC0)

In the second premise, the plan is still in state ready.

ready#('SNpi', sk; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG)

The reason for these two premises is as follows: In Fig. 1 the outgoing arc from `ready` is guarded. The guard is only satisfied, if an activate signal is sent to the plan by its respective parent. However, it is not guaranteed, that this signal will be sent. Therefore, `'SNpi'` may stay in the ready state indefinitely. Induction can be used to close this premise.

Proof continues with an activated `'SNpi'` plan as seen in node 4 in Figure 6. After the execution of another step, four further premises are generated. In the first premise, the plan is aborted, in the second completed. These goals can be closed easily. In the third premise, `'SNpi'` is still activated. We can solve this goal again with induction.

In the fourth proof obligation, marked as node 5 in Fig. 6, `'SNpi'` is suspended. We need a further temporal step here, resulting in two open premises, one, where the state remained suspended - a goal where again induction can be applied - and a second goal, where the plan becomes activated again. In this goal, induction is possible to the last state, where `'SNpi'` was activated, which is node 4 in Figure 6. Therefore, this proof is closed.

5.4 Rely guarantee approach

The original proof obligation is rather weak. It has only been verified, that the property holds, if the environment never changes the state of the `'SNpi'` slot of the *AS* variable. This has been a deliberate weakening of the proof obligation, to better illustrate the structure of an `asbru` proof in KIV.

Assume a trace, where the environment violates the environment assumption in the fifth TL step. Obviously, this trace is not allowed, so the Asbru procedures may show arbitrary behaviour within this trace. Especially, `'SNpi'` may violate the property in the first step. Therefore, a stronger version of the property is wanted.

Our proposed solution is generally known as the "rely guarantee" or "assumption commitment" approach. This has been proposed for example by [1]. We will present our implementation of this idea.

A first step towards a strengthened properties is the use of the **unless** operator. The **unless** operator allows us to force a system – the plan `'SNpi'` in that case

– to show desired behaviour up to the point, where unwanted behaviour in the environment can be noted. This can be done in the following way:

(: guarantee :)
([Frame AS, Tick] ∧ frame(AS', AS, 'SNpi' '))
unless (: negated rely :)
¬ AS"['SNpi'] = AS'['SNpi']

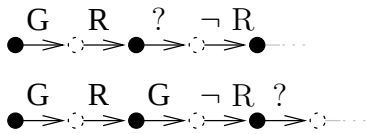


Fig. 7. Property violation with unless

However, we could further strengthening this. In our execution engine, system and environment transitions alternate. The **unless** as stated above, allows the upper trace in Fig. 7. As the environmental transition follows the system transition, the systems may again violate the property before the misconduct of the environment occurred. What would be wanted was the lower trace. This can be formulated in our example as

(: system configuration :)
AS['SNpi'] = inactive,
(: system description :)
[inactive#('SNpi', sk; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG)]
⊢ (: property :)
([Frame AS, Tick] ∧ frame(AS', AS, 'SNpi' '))
unless ((: repeated guarantee :)
[Frame AS, Tick] ∧ frame(AS', AS, 'SNpi' ')
(: negated rely :)
∧ ¬ AS"['SNpi'] = AS'['SNpi']

Looking closely at the presented example, one can see, that in this case, an even stronger property can be formulated. The plan `'SNpi'` should never violate the property, even if the environment assumption is violated. As `'SNpi'` does not require the *AS* variable for its progression, the proof should not be considerable different, once the environment assumption is omitted.

Also it is possible to omit the state of the *PC* variable. If an inactive plan does not get its consider signal, the procedure will terminate. However, with a terminating system, the property is trivially satisfied, which leads to the property below.

(: system description :)
[inactive#('SNpi', sk; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG)]
⊢ (: property :)
□ ([Frame AS, Tick]
∧ frame(AS', AS, 'SNpi' '));

6 Compositional verification of large hierarchies

Aim of this section is to provide a hands on approach on the modular verification of a medically relevant property. For this, we will take a look at an intention, stating *After axilla clearance, the axilla should not normally be irradiated*. We will first describe the property, then the expected problems when using symbolic execution, as presented in Sect. 5 and finally our proposed solution.

6.1 Guideline Background

The guideline used as a basis for our verification work is dealing with the treatment of breast cancer. The guideline is structured into eight chapters, where chapters one through six deal with the treatment of the disease in different phases. The verification work described here is done in the first chapter. This chapter deals with treatment of operable invasive breast cancer and DCIS, which is ductual carcinoma in situ.

Treatment may consist of surgery as well as chemotherapy or radiotherapy. For this paper, we are especially interested in surgery of the axilla and the follow up treatments of it.

6.2 Medical goal and translation

Our proof obligation is a medical property. Good practice medicine should ensure, that *after axilla clearance, the axilla is not normally irradiated*. This goal has been processed according to the translation patterns described in [6]. Result of this translation process was the finding, that *axilla clearance* should be associated with the Asbru plan *clearance of axilla node*, abbreviated to `'cand'`. It has been determined, that the irradiation of the axilla can be associated with the Asbru plan *primary radiotherapy of the axilla*, abbreviated to `'praoa'`.

According to the translation pattern, some starteventtrigger and some endeventtrigger have to be defined describing the interval of time, during which the desired behaviour – the non-activation of plan `'praoa'` – has to be observed. The starteventtrigger is defined as the time point, where plan `'cand'` is completed, the endeventtrigger is defined as the time of termination of the *dealing with axilla* plan (abb. `'dwa'`). The plan `'dwa'` has been chosen, because it is the first common parent of `'cand'` and `'praoa'`. The relevant section of the Asbru

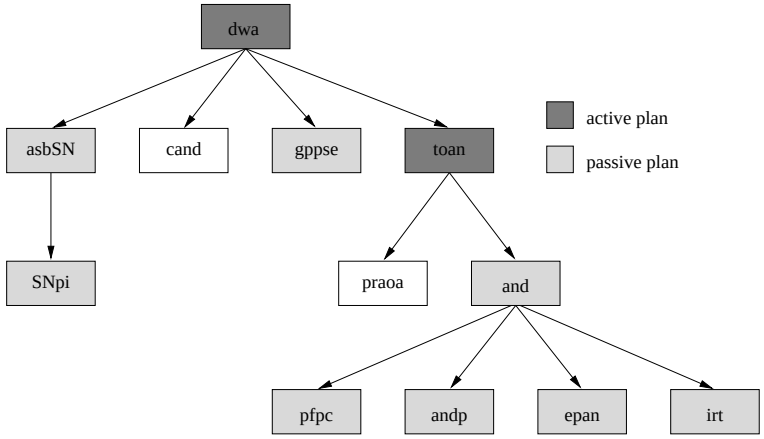


Fig. 8. Section of the Asbru plans of chapter 1

plan hierarchy can be seen in Figure 8, the proof obligation is written down below.

$$
\begin{aligned}
&[\text{inactive\#}(\text{`dwa'}, \text{`treat'}\\
&\qquad ; \text{Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG})],\\
&\text{Tick},\\
&\Box\ (\quad (\text{AS''})[\text{`dwa'}] = (\text{AS'})[\text{`dwa'}] \wedge (\text{AS''})[\text{`cand'}] = (\text{AS'})[\text{`cand'}]\\
&\quad \wedge\ \text{AS''}[\text{`praoa'}] = \text{AS'}[\text{`praoa'}] \wedge \text{AS''}[\text{`toan'}] = \text{AS'}[\text{`toan'}]\\
&\quad \wedge\ (\neg\ \text{Tick'} \rightarrow \text{PDH''}[\text{AC''}] = \text{PDH'}[\text{AC}]) \wedge \text{Tick''}\\
&\quad \wedge\ \text{suspended} \neq (\text{AS''})[\text{`treat'}]),\\
&\neg\ \text{StartEventTrigger} \wedge \neg\ \text{EndEventTrigger},\\
&\text{AS}[\text{`praoa'}] = \text{inactive},\\
&\Box\ (\text{if } (\text{AS}[\text{`cand'}] \neq \text{completed} \wedge \text{AS'}[\text{`cand'}] = \text{completed})\\
&\quad \text{then StartEventTrigger''}\\
&\quad \text{else } (\text{StartEventTrigger'} \leftrightarrow \text{StartEventTrigger''})),\\
&\Box\ (\text{if } (\quad \text{StartEventTrigger} \wedge \text{AS}[\text{`dwa'}] \neq \text{completed}\\
&\qquad \text{AS'}[\text{`dwa'}] = \text{completed})\\
&\quad \text{then EndEventTrigger''}\\
&\quad \text{else } (\text{EndEventTrigger'} \leftrightarrow \text{EndEventTrigger''})),\\
&\Box\ (\text{if } (\text{StartEventTrigger} \wedge \neg\ \text{EndEventTrigger''} \wedge \text{activated} = \text{AS}[\text{`praoa'}])\\
&\quad \text{then GoalFailed''}\\
&\quad \text{else } (\text{GoalFailed''} \leftrightarrow \text{GoalFailed'})),\\
\vdash\ &\Box\ (\text{StartEventTrigger} \rightarrow \text{activated} \neq \text{AS}[\text{`praoa'}] \vee \text{EndEventTrigger})
\end{aligned}
$$

In our work, we distinguish between active and passive plans. These terms are relative to the verification of one property. Plans, designated as active somehow contribute to the fulfilment of the property. This can be, because their plan state

is relevant for the evaluation of the property, they are parents of such plans or guarantee certain environmental conditions by their filter and setup conditions.

As can be seen in Fig. 8, plans directly relevant for the evaluation of the property have a white background. Other active plans are shaded with a dark grey, passive plans in a light grey. All in all, the section of the hierarchy contains 12 plans in four hierarchical layers.

This is a rather large hierarchy. Although the use of the verification technique, described in Section 5 is possible, it is not advised. The indeterminism inherent in some of the plans would lead to a very large proof. Instead it is suggested to abstract the subplans of `'dwa'`. For this, we propose to use the rely guarantee approach already sketched in Section 5.4.

6.3 Proof idea

Verification complexity grows with the number of indeterministic controls, like the any-order control, and execution speed falls with the number of $\|_s$ operators. Therefore, it is recommended to use a different approach towards verification of such hierarchies. An approach reducing the overall number of parallel running plans and hiding the indeterminism.

One example, how this works, is in the hierarchy in Fig. 8 with the plan *axilla node dissection*, abbreviated `'and'`. It might be important for the property described in Sect. 6.2, if plan `'and'` aborts or completes its execution. However, it seems to be unimportant if, for example, its subplan `'pfpc'` or its subplan `'irt'` was responsible for the abortion of `'and'`. Therefore we aim to have an abstraction of plan `'and'`. This abstraction should start running, when the procedure representing `'and'` is spawned. The abstraction should indeterministically complete or abort at some point. During its execution, this abstraction should maintain basic properties like the non-interference with *AS* slots of plans outside the `'and'` subtree.

In a similar way, we seek abstraction properties for all the plans but `'dwa'`. For the passive plans `'gppse'` and `'asbSN'`, these properties will be similar to that of `'and'`. The property for `'toan'`should be, that its child `'praoa'` is not activated as long as `'toan'` has not been activated on its own. For `'cand'` it is necessary to know, that after `'cand'`s termination, `'cand'` will no longer change its state.

We will now concentrate on the sub hierarchy controlled by plan `'toan'`. We aim to verify, that plan `'praoa'` cannot get activated, as long as `'toan'` gets activated itself. We further want to show, that `'toan'` always completes, if the parent `'dwa'` does not terminate. For this, we have to know, that the necessary child `'praoa'` cannot abort or reject.

In the following section, we will show how to verify the abstraction for plan `'praoa'`, afterwards how to combine this abstraction with one for plan `'dwa'`.

Finally, we will show, how these three proofs can be used in the verification of the abstraction of plan `'toan'`.

6.4 Verifying simple RG Properties

We try to verify a property of the "plan primary radiotherapy of the axilla", `'praoa'`. The property is named `praoa-non-abort`. With this property, we abstract the behaviour of `'praoa'` in a way, that the plan may not abort or reject, as long as the parent does not abort. Also, it is necessary to state, that the plan will not be activated unless it receives its `activate` signal from its parent `'toan'`.

We also verify a technical property, that slots in the AS variable other than `'praoa'` are left alone. This property can be guaranteed without any assumption to the environment, while the other part of the property needs two assumptions, the parent not aborting and the state of the AS slot of `'praoa'` not to be changed by the environment. For technical reasons, the property has to be written down in a certain syntactical form:

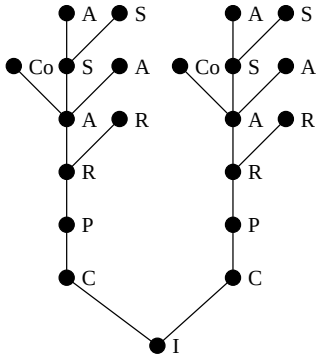
praoa-non-abort:
⊢ (: system description :)
[inactive#('praoa', sk; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG)]
(: current state :)
→ AS['praoa'] = inactive
(: first part of property :)
→ □ ([FRM AS, Tick] and frame(AS, AS', 'praoa' '))
(: guarantee of second part of property :)
∧ ¬ (AS['praoa'] = aborted ∧ not AS['praoa'] = rejected)
(: repeated guarantee of second part of property :)
unless (¬ AS['praoa'] = aborted
∧ ¬ AS['praoa'] = rejected
(: negated rely of second part of property :)
∧ (¬ AS"['praoa'] = AS'['praoa']
∨ terminatedP(AS[sk])))

After start of the verification, the proof obligation is automatically rewritten to

praoa-non-abort:
(: system description :)
[inactive#('praoa', sk; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG)],
(: current state :)
AS['praoa'] = inactive
⊢ (: first part of property :)
□ ([FRM AS, Tick] and frame(AS, AS', 'praoa' '))

(: guarantee of second part of property :)
$\wedge \neg$ (AS[‘praoa’] = aborted $\wedge$ not AS[‘praoa’] = rejected)
(: repeated guarantee of second part of property :)
unless ($\neg$ AS[‘praoa’] = aborted
$\wedge \neg$ AS[‘praoa’] = rejected
(: negated rely of second part of property :)
$\wedge$ ($\neg$ AS”[‘praoa’] = AS’[‘praoa’]
$\vee$ terminatedP(AS[sk])))

Which falls into two separate proof obligations, splitting the property at the leading conjunction in the succedent. Both proofs will be described separately.



Verification Verification of the first part of the property very closely resembles verification of the property in Section 5.

Verification of the second part of the proof obligation `praoa-non-abort` is again similar to the verification done in Section 5. With some specific knowledge about the plan `‘praoa’` regarding its termination behaviour, some side arms no longer show up. Due to the non existing filter and abort conditions and the fact, that the parent of `‘praoa’` may not abort – as stated by the environment assumption – the corresponding cases do not have to be considered.

Combining TL properties It is sometimes necessary to combine different TL properties in a way that they can be rewritten as a single property. Take for example the previous proof, where it had to be verified, that *AS* slots other than `‘praoa’` are left alone, as well as `‘praoa’` will not abort. In such cases, sometimes the proving of sub properties can take considerable more effort than in the current case. It is not too unlikely, that a verifier does not get the property right in the first attempt. Going along the lines of Murphys law, usually this is

discovered when working on the last open premise. In that case, much of the previous proof effort may be lost or may be subject to replay. We therefore suggest to decouple the properties.

Assuming, the property is written down as

```
   System
→     state_for_prop_1 ∧ state_for_prop_2
   →     prop1
      ∧ prop2
```

we suggest to formulate two properties

```
   System
→     state_for_prop_1
   → prop1
```

and

```
   System
→     state_for_prop_2
   → prop2
```

and use them as rewriting rules to prove the original property. That way, if errors are for example found in `state_for_prop_2` or even within `prop2` only a partial proof is affected.

6.5 Combining rely guarantee properties

Although the abstraction of Asbru plans may allow for faster application of induction, the effort for verification stays virtually the same. The reason for this is, that complexity is not mainly based on the programs themselves, but on their number. Therefore the ultimate goal is not the abstraction of plans but the reduction of parallel operators.

It is possible to combine different RG properties that are executed in parallel into one single RG property. The result of such combinations is a great reduction in the number of generated goals after a temporal logic step. It is obvious, that two parallel **unless** formulas can generate more indeterminism than one.

The suggested procedure is to determine a RG property which is a generalisation of the synchronous parallel combination of RG properties and verify in a separate proof, that the generalisation is indeed valid. This property can then be used within the main proof to eliminate parallel operators, thus increasing the performance of the TL step and reducing complexity by reducing the indeterminism.

There has been found a compositional theorem to reduce the combination of two RG properties into one to a mere predicate logic one. This holds for both, the combination of unless properties and the combination of always properties. There is no automated support for the application of this theorem, yet. Therefore, application of this theorem can only be done manually outside the KIV system. Alternatively, it is possible to prove the abstraction to be correct in a temporal logic proof.

In many cases the resulting rely is merely a combination of both single relies and the same holds for the guarantees. For example in the `praoa-non-abort` proof fusing two always properties the guarantees for both properties are

□ ([FRM AS, Tick]
∧ AS = AS'[‘and’, AS[‘and’]][‘andp’, AS[‘andp’]]
[‘epan’, AS[‘epan’]][‘irt’, AS[‘irt’]]
[‘pfpc’, AS[‘pfpc’]])

and

□ ([FRM AS, Tick] ∧ AS' = AS[‘praoa’, (AS')[‘praoa’]])

the resulting guarantee is

□ ([FRM AS, Tick]
∧ AS = AS'[‘and’, AS[‘and’]][‘andp’, AS[‘andp’]]
[‘epan’, AS[‘epan’]][‘irt’, AS[‘irt’]]
[‘pfpc’, AS[‘pfpc’]][‘praoa’, (AS')[‘praoa’]])

The combination is something quite similar as the conjunction of both properties, however, the equation for the Asbru state variable makes things a little bit more complex here. Both processes are allowed to modify their *AS* slots independently. This is stated inversely, where it is said, that all other slots are left unchanged. Simply combining the properties would lead to a situation, where no slot in the *AS* could be changed.

6.6 Using rely guarantee properties

The properties described in the section 6.4 can be used to reduce the complexity of other proofs. It is obvious, that the complexity grows with every additional subplan that has to be considered for the verification of the property. With the rely guarantee approach, it is possible to abstract a complete plan hierarchy of several levels to a single, flat TL formula. Consider the previous sections. It is not hard to see, this technique can also be applied to more complex plans.

This section will describe how to use the abstracted properties with the verification of a RG property. For this, we will take a sub property to be used in the

proof of the medical goal described in Section 6.2. This proof obligation is called `toan-20`, defined in specification `treatment-axilla`.

Our proof obligation is

$$
\begin{array}{l}
\vdash \\
\quad \text{(: system description :)} \\
\quad [\text{inactive\#}(\text{`toan'}, \text{sk; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG})] \\
\qquad \text{(: system configuration :)} \\
\rightarrow (\quad \text{AS[`toan'] = inactive and PC[`toan'] .consider} \\
\qquad \text{(: guarantee :)} \\
\quad \rightarrow (\quad \text{(not terminatedP(AS[`toan']) and not laststep)} \\
\quad \text{unless (} \quad \text{(: regular exit :)} \\
\qquad\qquad \text{AS[`toan'] = completed} \\
\qquad\qquad \text{(: negated rely :)} \\
\qquad\quad \vee\ \text{AS[sk] = suspended} \\
\qquad\quad \vee\ \text{terminatedP(AS[sk])} \\
\qquad\quad \vee\ \text{(AS')[`toan'] } \neq \text{ (AS")[`toan']} \\
\qquad\quad \vee\ \text{(AS')[`praoa'] } \neq \text{ (AS")[`praoa']))}
\end{array}
$$

For a better understanding, we would like to further explain this property.

Within the system description, it can be seen, that the top level plan for this property is `'toan'`, which is the above mentioned treatment-of-axilla-node. We have left the parent anonymous, so should this plan ever become part of another plan hierarchy with another parent, this property might be reused.

As can be seen, a system configuration is given within this property. The reason for this is not, that the plan requires it for its execution, but that the property might be violated initially. Should the system start with the Asbru state of `'toan'` being set to terminated, the guarantee would be violated initially.

The guarantee verified here, is, that the plan `'toan'` will never abort or reject and thus either run infinitely long or complete its execution. There is a slight deviation of the "pure" RG scheme. We allow for a regular exit without the violation of the environment assumption here, that is, we consider our property to hold, if the plan completes, which is not a violation of the rely.

The rely is, that the parent of plan `'toan'` is not suspended or terminated. The latter one is necessary, as every plan immediately aborts, once it can be established that its parent is either aborted or completed. Also, we need the environment not to touch the Asbru state of `'toan'` itself and one of its children, `'praoa'` which is the plan primary-radiotherapy-of-axilla. In case, the Asbru state of that child is changed to aborted or rejected by the environment, `'toan'` might abort, because `'praoa'` is a necessary child of `'toan'`.

The property is symbolically executed in the manner described in section 6.4 to the point, where the subplans of `'toan'` are spawned. After the spawning, the subplans are rewritten by their respective RG properties.

```
[var PC0 = init[‘praoa’ ’ + ‘and’ ’, true, false, false]
  in begin
      activated#(‘toan’, sk, [], ‘praoa’ ’ + ‘and’ ’, [], 1
                  ; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG, PC0
                  ) ‖s
      begin
          inactive#(‘praoa’, ‘toan’
                      ; Tick, Patient, PDH, VH, AC, ASH, AS, PC0, EAGG
                      ) ‖s
          inactive#(‘and’, ‘toan’
                      ; Tick, Patient, PDH, VH, AC, ASH, AS, PC0, EAGG
                      )
      end
  end
]
```

```
[var PC0 = init[‘praoa’ ’ + ‘and’ ’, true, false, false]
  in begin
        activated#(‘toan’, sk, [], ‘praoa’ ’ + ‘and’ ’, [], 1
                    ; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG, PC0
                    ) ‖s
        begin
            [PL (   □ (   [FRM AS, Tick]
                        ∧ frame(AS, AS’, ‘praoa’ ’))
                  ∧   (   AS[‘praoa’] ≠ aborted
                        ∧ rejected ≠ AS[‘praoa’])
                    unless (   aborted = AS[‘praoa’]
                             ∧ rejected ≠ AS[‘praoa’]
                             ∧ (   (AS”)[‘praoa’] ≠ (AS’)[‘praoa’]
                                 ∨ terminatedP(AS[‘toan’])))))
            ] ‖s
            [PL □ (   [FRM AS, Tick]
                    ∧ frame(AS, AS’, ‘and’ + ‘pfpc’ + ‘andp’ + ‘epan’ + ‘irt’ ’))
            ]
        end
  end
]
```

After this rewrite step, the properties are linked together by a lemma like those described in section 6.5. The outcome of this lemma application in combination with some simplification is

```
[var PC0 = init[‘praoa’ ’ + ‘and’ ’, true, false, false]
  in begin
```

```
        activated#('toan', sk, [], 'praoa' ' + 'and' ', [], 1
                    ; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG, PC0
                    ) ||s
        [PL (   □ (   [FRM AS, Tick]
                    ∧ AS' = (AS['and', (AS')['and']]['andp', (AS')['andp']]
                                ['praoa', (AS')['praoa']]['epan', (AS')['epan']]
                                ['irt', (AS')['irt']]['pfpc', (AS')['pfpc']]
                    ))
              ∧ (   aborted ≠ AS['praoa']
                  ∧ rejected ≠ AS['praoa']))
              unless (   aborted ≠ AS['praoa']
                       ∧ rejected ≠ AS['praoa']
                       ∧ (   (AS")['praoa'] ≠ (AS')['praoa']
                           ∨ terminatedP(AS['toan'])))
              ]
        end
]
```

As can be seen here, the property is a combination of unless and always properties, putting every sub property holding unconditionally into the always part and only those necessary into the unless part. Our experience shows, this form of formulation does reduce proof effort and increase execution speed, so in principle we have here a trade off between more complex property formulation and more efficient verification of the main proof.

After this lemma application, the knowledge about the states of `'and'` and `'praoa'` is weakened. As the RG properties abstract from the control flow of the asbru plans, the knowledge, that `'and'` and `'praoa'` are inactive here is no longer important. For `'praoa'` the only important thing is, that it is neither aborted nor rejected initially, which is guaranteed by the unless formula. As there is no active contribution of `'and'` all knowledge can be discarded.

This drastic procedure is only advisable, as long as the property for the subplan does not contribute in an active sense or the active contribution can be guaranteed by the RG property. Otherwise, it could be crucial to know, that a certain subplan is not active, not completed or not aborted. There is however no generic possibility to foresee all possible generalisations. Formulation of those is a creative step.

This is the goal before application of the step rule

```
N = N" + 1 until Boolvar,
[var PC0 = pc
  in begin
        activated#('toan', sk, skx, sky, [], 1;
                    Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG, PC0
                    ) ||s
        [PL (   □ (   [FRM AS, Tick]
```

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \wedge\ \text{AS'} = (\text{AS}[\text{`and'}, (\text{AS'})[\text{`and'}]][\text{`andp'}, (\text{AS'})[\text{`andp'}]] \\
\qquad\qquad\qquad\qquad\qquad\qquad [\text{`epan'}, (\text{AS'})[\text{`epan'}]][\text{`irt'}, (\text{AS'})[\text{`irt'}]] \\
\qquad\qquad\qquad\qquad\qquad\qquad [\text{`pfpc'}, (\text{AS'})[\text{`pfpc'}]][\text{`praoa'}, (\text{AS'})[\text{`praoa'}]] \\
\qquad\qquad\qquad\qquad)) \\
\qquad\qquad\qquad \wedge \quad (\quad \text{AS}[\text{`praoa'}] \neq \text{aborted} \\
\qquad\qquad\qquad\qquad\ \wedge \text{AS}[\text{`praoa'}] \neq \text{rejected}) \\
\qquad\qquad\qquad \text{unless } (\quad \text{AS}[\text{`praoa'}] \neq \text{aborted} \\
\qquad\qquad\qquad\qquad\ \wedge \text{AS}[\text{`praoa'}] \neq \text{rejected} \\
\qquad\qquad\qquad\qquad\ \wedge (\quad (\text{AS''})[\text{`praoa'}] \neq (\text{AS'})[\text{`praoa'}] \\
\qquad\qquad\qquad\qquad\qquad\ \vee \text{terminatedP}(\text{AS}[\text{`toan'}])))) \\
\qquad\qquad] \\
\qquad\ \text{end} \\
\quad], \\
\quad \text{INDHYP-nat}(((((\text{N} + 1) + 1) + 1) + 1) + 1), \\
\quad \text{AS}[\text{`praoa'}] \neq \text{completed}, \text{AS}[\text{`toan'}] = \text{activated}, \\
\quad \text{AS}[\text{`praoa'}] \neq \text{aborted}, \text{AS}[\text{`praoa'}] \neq \text{rejected} \\
\vdash \quad (\quad \text{AS}[\text{`toan'}] \neq \text{rejected} \\
\qquad\ \wedge \text{AS}[\text{`toan'}] \neq \text{aborted} \wedge \text{AS}[\text{`toan'}] \neq \text{completed} \\
\qquad\ \wedge \neg\ \text{laststep}) \\
\qquad \text{unless } (\quad \text{AS}[\text{`toan'}] = \text{completed} \vee \text{AS}[\text{sk}] = \text{suspended} \\
\qquad\qquad\ \vee \text{terminatedP}(\text{AS}[\text{sk}]) \vee (\text{AS'})[\text{`toan'}] \neq (\text{AS''})[\text{`toan'}] \\
\qquad\qquad\ \vee (\text{AS'})[\text{`praoa'}] \neq (\text{AS''})[\text{`praoa'}]
\end{array}
$$

Looking closely, one will notice the plan lists describing the running and evaluated subplans of `'toan'` have been abstracted to generic values. Although this is not strictly necessary, it has been found to reduce the proof effort by a factor of three to four. Although this abstraction leads to some more goals after the initial step, the fact that all goals without terminating plans can be closed by induction makes up for this.

After applying a temporal logic step, three types of proof obligations are generated. One type, where predicate logic reasoning is necessary, one type where induction can be used. In a third type of goals, the system description changed, because `'toan'` has completed. In these goals, further temporal logic reasoning is necessary.

In some cases it can be necessary to abstract from the lists of the activated procedure again after it has been found, that further steps are necessary. Although it can never happen, that plan names are inserted out of nowhere, it can happen, that plan names move from one list to another.

Plans can only move from the right handed lists to the left handed lists. They can also drop from the lists. This should be considered, before further steps.

6.7 Generic properties

As mentioned above, proof for certain properties always follows a certain scheme. The resemblance is so high, that verification efforts can be combined. After some generalisation proof obligations can be obtained and verified, sufficient to satisfy multiple more specialised proof obligations. Such generic properties, to be reused for multiple proofs mark one of the keys to significantly reduce the overall proof effort.

To come up with such generic properties, it is necessary, to find out the essential structural part of the original property. In the above mentioned case of `'praoa'` and `'SNpi'`, it is only important to know, that `'praoa'` is a user-performed plan with no children of its own, the same as the `'SNpi'` plan in Section 5.

Therefore, the following property is defined

user-sane:
(: system description :)
$\vdash$ [inactive#(sk, sk0; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG)]
(: requirements to the plan :)
$\rightarrow$ (asbru(sk) .type = user $\wedge$ asbru(sk) .subplans = []
(: property :)
$\rightarrow \Box$ ([FRM AS, Tick] and frame(AS, AS', 'praoa' '))

Verification of this property is almost identical to the corresponding property in Section 5.4. With this property, among many others, the first proof obligation `praoa-non-abort` can be verified.

6.8 Conclusion

Our experience comparing the head on approach with symbolic execution and the abstraction (also called proof decomposition) shows great potential with the latter technique. Symbolic execution with larger plan hierarchies leads to unacceptable proof sizes. In contrast, the proof decomposition technique leads to proofs of almost constant size, where only the size of the proofs of the auxiliary lemmas vary. However, for those the technique can be applied recursively to reduce effort for their proofs.

Right now there has been gathered much more experience with the direct symbolic execution than the proof decomposition. Therefore the limits of this approach have not yet been found. As of now, it seems as the use of this proof technique is not as straight forward as the symbolic execution, meaning that more knowledge and experience is required by the verifier. Furthermore it is not yet as well automated as the symbolic execution and it seems that some of the predicate logic goals are inherently difficult to automate with the necessity of instantiation of quantifiers or the application of lemmas that - on first sight - can not be applied automatically.

References

1. M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 3 Nov 1995.
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. 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/`.
5. A. Seyfang, S. Miksch, and M. Marcos. Combining diagnosis and treatment using Asbru. *International Journal of Medical Informatics*, 68((1-3)):49–57, 2002.
6. Ruud Stegers. From natural language to formal proof goal: Structured goal formalisation applied to medical guidelines. Master's thesis, Vrije Universiteit, Amsterdam, 2006.