IST-FP6-508794

PROTOCURE II

Integrating formal methods in the development process of medical guidelines and protocols

Specific Targeted Research Project
Information Society Technologies

Complementary material to D4.2b Improved Verification System

Due date of deliverable: 31 October 2005
Actual submission date: 25 October 2005

Start date of project: 1 January 2004 **Duration:** 30 months

Organisation name of lead contractor: Universitat Jaume I de Castellón

Revision 1

Project co-funded by the European Commissions within the Sixth Framework Programme(2002-2006)		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Complementary material to Deliverable D4.2b: Improved Verification System (prototype)

Jonathan Schmitt, Michael Balser, Wolfgang Reif
University of Augsburg

October 31, 2005

Abstract

In previous work-packages, techniques were proposed to model medical guidelines in a notation which is more structured, and especially more precise than informal text. The informal text of a medical guideline is annotated in MHB and afterwards translated to the Asbru language. Asbru is a plan oriented language especially designed for modelling medical guidelines. The semantics of Asbru has been formally defined making it possible to analyze Asbru plans with formal methods. In the first Protocure project, basic support for the verification of Asbru plans has been added to the interactive theorem prover KIV. The support has been restricted to a subset of the Asbru language and proofs were difficult to construct. The goal of this work-package has been to improve the support for Asbru in KIV such that arbitrary temporal properties for any Asbru plan can be verified. Special care has been taken to ensure that proofs can be decomposed. This ensures that large Asbru plans can be verified in KIV.

This document complements the prototype and especially contains a tutorial on how to verify properties of Asbru plans in KIV.

Contents

1 Introduction **5**

2 Asbru semantics **5**

3 Proof obligations **7**

3.1 System description 7

3.2 System configuration 8

3.2.1 Asbru state 8

3.2.2 Variable, patient and patient data 9

3.2.3 Environment aggregation and plan communication 10

3.3 Environment assumption 10

4 Symbolic Execution of Asbru **11**

4.1 Normal form 11

4.2 Rewriting temporal operators to normal form 12

4.3 Executing an overall step 13

4.4 Executing Asbru 14

4.5 Simplifier rules 14

4.5.1 Simplifier rules for Asbru plans 14

4.5.2 Simplifier rules for Asbru semantics 15

4.6 Induction 17

4.7 Heuristics 18

5 Verifying a single Asbru plan **18**

5.1 Induction 19

5.2 First step 19

5.3 Further steps 19

6 Verifying a small hierarchy of Asbru plans **20**

6.1 Induction 20

6.2 First step 21

6.3 Further steps 21

6.4 Activating the sub plan 22

6.5 Environment interaction 22

7 Compositional verification of large hierarchies **23**

7.1 Rely guarantee in theory 23

7.2 Verification of basic rely guarantee properties 24
7.3 Combining rely guarantee properties 25
7.4 Using rely guarantee properties 27
7.5 Conclusion 33

1 Introduction

Before being able to verify properties over systems in a given non-standard representation like Asbru, it is necessary to clearly define the desired behaviour 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 of Asbru Light had been revised to match the needs of the Protocure II case study breast cancer, it was necessary to implement it in KIV. As the breast cancer Asbru plans were not available until recently, it has been decided to take the Protocure I case study dealing with jaundice in newborns and test the implementation with it.

As the maturity of the KIV TL implementation has been greatly increased outside the Protocure II project, 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. Instead, the effort was invested to specify Asbru instead of implementing it.

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 semantics to proof properties.

2 Asbru semantics

The Asbru semantics are described in detail in [2]. The Implementation of the semantic in the KIV system is described in [3]. 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.

In principle, there are two alternatives to implement the Asbru semantics in KIV.

The first possibility is to hard code a data structure for Asbru plans in the

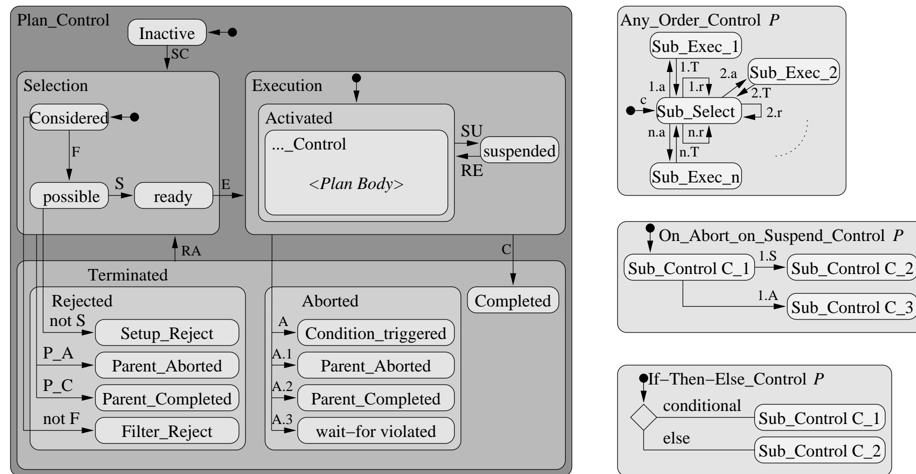

Figure 1: State transitions of Asbru plans

programming language of KIV. It is questionable to what extend already existing KIV features like the simplifier or the correctness management would work or how much effort it would take to make them work again.

Therefore, we opted for the second possibility, where Asbru plans are algebraic specified KIV data structures. The execution of the plans has also been specified in the means of parallel programs. This turned out to be a good choice when performing work for the living proofs prototype. There especially the smooth integration into the correctness management came in handy.

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 1. All transitions within this statechart are guarded, where the guards denoted `SC`, `RA` and `E` denote 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 denotate conditions which define the behaviour of the Asbru plan. An individual condition is 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 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 itself

This behaviour of Asbru plans is defined within the specification `Asbru`, where the statechart 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 always be confronted with the syntactical same program.

Usually, part of the status of a program is somewhat hidden within a program counter. With the choice of recursive one-step terminating programs, this is not possible. Instead, the implemented Asbru programs are designed similar to statecharts, where state and control are strictly separated. The 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 current 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 defined in specification `environment-aggregation`.

3 Proof obligations

A proof obligation consists of two parts. For example such a proof obligation or property describes, that an Asbru plan adheres to an indicator. One of the parts of the property describes the behaviour of the Asbru plan hierarchy, the other one the behaviour, that is expected. 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 singular Asbru plan, therefore plan hierarchies are represented by a number of parallel running plans.

As the execution of Asbru plans is somewhat generic, all plantypes are represented by the same program. Plans on the sequent vary only in the parameters with which they are called. To be precise, the variation extends only to two parameters. These are the name of the plan and the name of the parent. With the first parameter, the program can look up the definition of the plans conditions, its subplans and so on. With the second, the plan can check if the parent is still running or the plan has to react to the termination or suspension of its father.

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 state variables, as the state variables contain the data for all Asbru plans. Each plan then selects values relevant for itself. The asbru plans have been carefully designed such that concurrent write access to the same data

fields cannot happen.

For the user it is in principle not necessary to care about the details of the procedures currently running in the sequent. All of these procedures should look alike. The are designed in a way that they are self recursive and each iteration terminates within one step. Therefore, the user only has to keep track of the names of the plans that are currently represented by procedures on the sequent. Their state cannot be read out of the programs but has to be read by the predicate logic representation of the overall state.

3.2 System configuration

As described in section 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.

3.2.1 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. An entry may look this way:

```
considered
activated([], ‘pob’ ’, [], 1)
```

The first part of the example represents the state of a considered Asbru plan, while the second represents an active plan, a state which is further elaborated. The three lists that accompany an active plan depict the subplans of this plan and denote to some extend the status, in which they are. The first list denotes all subplans that are currently active themselves, the second list those that are in the evaluation phase and the third list denotes all subplans, yet in inactive state. Plans that are terminated do not appear in those lists. The number is a counter, which is necessary when dealing with cyclical plans. For all other plan types, it is ignored. Cyclical plans allow to specify, that they have to be completed for n times, before the overall plan is seen as completed. To keep track of the times, the plan completed, this counter is used.

The status itself is not significant without the knowledge of the name of the plan. The planname however is used as a key to select the status out of the Asbru state. There are two ways how to denote 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['ob', activated([], 'pob' ', [], 1)]
        ['pob', considered]

vs. implicit
AS['ob']  = activated([], 'pob' ', [], 1)
AS['pob'] = considered
```

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 treated within this proof but may still be running concurrently. This concept of 'invisible' plans is especially important when dealing with proof decomposition. The basic Asbru state is then updated by one update per running plan. The concept proved especially hard to handle when it was necessary to generalise before the use of induction. Therefore it was abandoned for the implicit denotation.

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.

3.2.2 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 defined time points measurements to the patient may be made and therefore the data been transferred to the 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.

This model allows for further extensions, where malfunctions of gauges can be incorporated.

As with the Asbru state, there is an explicit and implicit denotation, with only the implicit being used. The example denotes, 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.

3.2.3 Environment aggregation and plan communication

The environment aggregation summarises all signals that may be sent to the plans by the environment, that is, 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 technically the signals manual activation or overridability, which influence the evaluation of conditions.

Each entry in the environment aggregation is linked to a plan via its key and consists of two sets of six signals to the conditions, 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 would be written on the sequent as

```
PC[‘pob’] .consider
```

As with the other data structures, there is a more explicit denotation of updates, in this case written down like this

```
PC[asbru(‘ob’) .subplans, false, false, false]
```

which is an update of all signals, sent by the parent to its child.

There is no history of signals, environmental or internal, because there is no need to incorporate them. Should properties be defined, that depend on histories of signals, signal histories could be included.

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:

```
\G (    AS’’[‘ob’]  = AS’[‘ob’]
    and AS’’[‘pob’] = AS’[‘ob’])
```

A weaker assumption 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

```
\G not suspendedP(AS''[`ob'])
```

An example for the second case is the logging of the Asbru state history. As the plans only change the Asbru state, not the history, this logging has to be done by the environment.

```
\G (    ASH''[AC''][`ob']  = AS'[`ob']
    and ASH''[AC''][`pob'] = AS'[`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.

```
\G (ASH[AC][`ob'] = ready -> \F EAGG[`ob'] .consider)}
```

4 Symbolic Execution of Asbru

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

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

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

$$\textit{name:} \quad \varphi \quad \leftrightarrow \quad \psi \; .$$

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

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

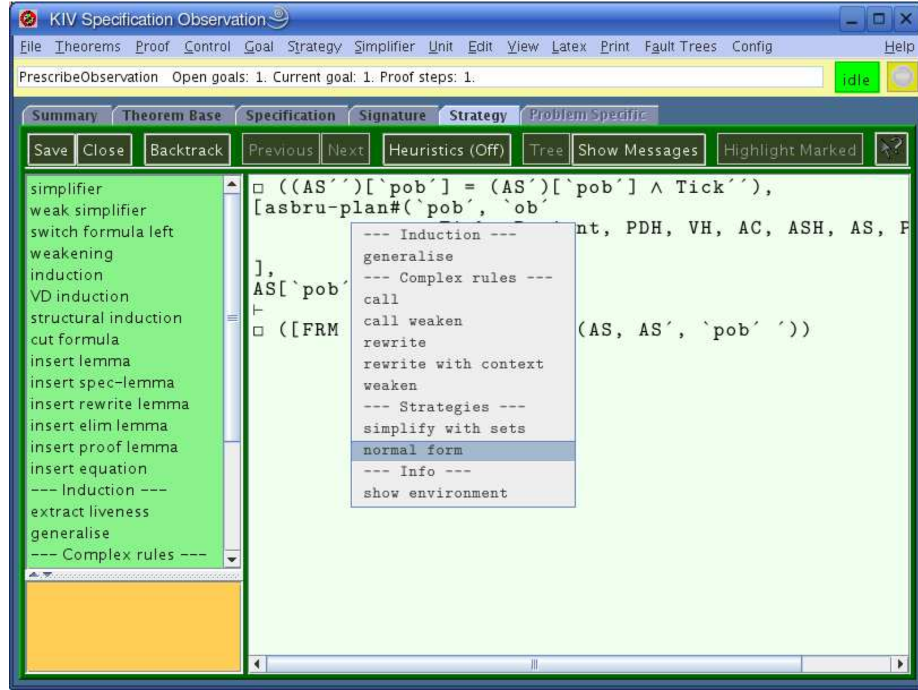

Figure 2: Rewriting sub-formulas to normal form

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 $\vec{X}_i$ which occur in both τ_i and φ_i. The general pattern to separate the first transitions of a given temporal formula is

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

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 [1]. To automatically rewrite the formula, select an arbitrary sub-formula and apply rule *normal form* as shown in Figure 2.

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

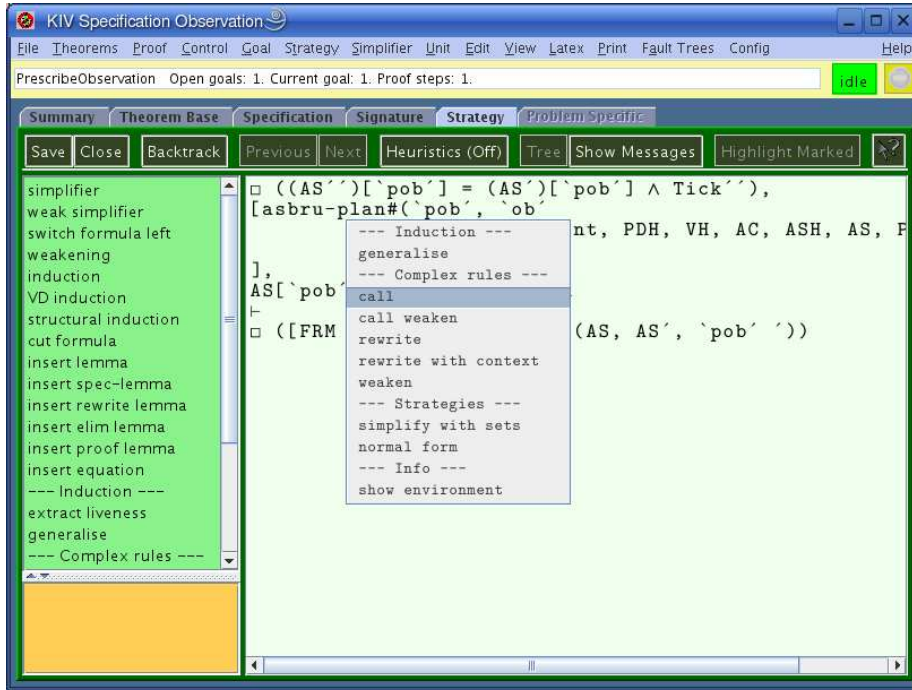

Figure 3: Rewriting the top-level operator

Figure 3).

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 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.)

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

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

$$\frac{\varphi, \Gamma \vdash \Delta \quad \psi, \Gamma \vdash \Delta}{\varphi \vee \psi, \Gamma \vdash \Delta} \; dis\ l \qquad \frac{\varphi[^{X_0}/_X], \Gamma \vdash \Delta}{\exists\ X.\ \varphi, \Gamma \vdash \Delta} \; ex\ l$$

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

$$\frac{\tau[^{X,X,X}/_{A,A',A''}] \vdash}{\tau, \mathbf{last} \vdash} \; lst \qquad \frac{\tau[^{X_1,X_2,A}/_{A,A',A''}], \varphi}{\tau, \circ\ \varphi \vdash} \; stp$$

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

Table 2: Rules for executing an overall step

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 normalizing 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 normalize those conditionals for which the conditions cannot be decided automatically.

4.5 Simplifier rules

4.5.1 Simplifier rules for Asbru plans

Axiom `ob-def` defines plan Observation

```
ob-def:
   asbru(‘ob’)
= mk-asbru-def
  (filter_condition,
```

```
mk-aasbruc
(mk-acond
 (lambda pdh, vh, ash, as, asbru-clock.
   pdf[ac][‘bilirubin’] .val = observation),
 false, false),
suspend_condition, resume_condition,
mk-aasbruc
(mk-acond
 (lambda pdh, vh, ash, as, asbru-clock.
   not pdh[ac][‘bilirubin’] .val = observation),
 false, false),
complete_condition, sequential, false, ‘pob’ ’,
wait-for-n(1, ‘pob’ ’), false)
```

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.

```
ob-filter:      asbru(‘ob’).filter = filter_condition;
ob-setup:       asbru(‘ob’).setup = mk-aasbruc(
                     mk-acond(
                       lambda pdh, vh, ash, as, asbru-clock.
                          pdh[ac][‘bilirubin’] .val = observation),
                       false, false);
ob-suspend:     asbru(‘ob’).suspend = suspend_condition;
ob-reactivate: asbru(‘ob’).reactivate = resume_condition;
ob-abort:       asbru(‘ob’).abort = mk-aasbruc(
                     mk-acond(
                       lambda pdh, vh, ash, as, asbru-clock.
                          not pd[‘bilirubin’].pdvalue.val = observation),
                       false, false);
ob-complete:    asbru(‘ob’).complete = complete_condition;
ob-type:        asbru(‘ob’).type = sequential;
ob-retry:       asbru(‘ob’).retry = false;
ob-subplans:    asbru(‘ob’).subplans = ‘pob’ ’;
ob-waitfor:     asbru(‘ob’).waitfor = wait-for-n(1, ‘pob’ ’);
ob-opt-wf:      asbru(‘ob’).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.

4.5.2 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 presence of a contradiction within the sequent 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

```
|- ps = inactive -> (ps = considered <-> false)
```

Precondition of this rule is, that a plan state variable is assigned the value 'inactive'. If this fact is written down on the sequent, than 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, than 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.

```
|- sk < sk1 -> (sk1 ' + sk ' = sk ' + sk1 ')
|- 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 sub-formulas, 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 (which is, in general, not enough).

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 abovementioned 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 |- (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 |- ((* sk1) ' + (* sk) ' = sk ' + sk1 ')
```

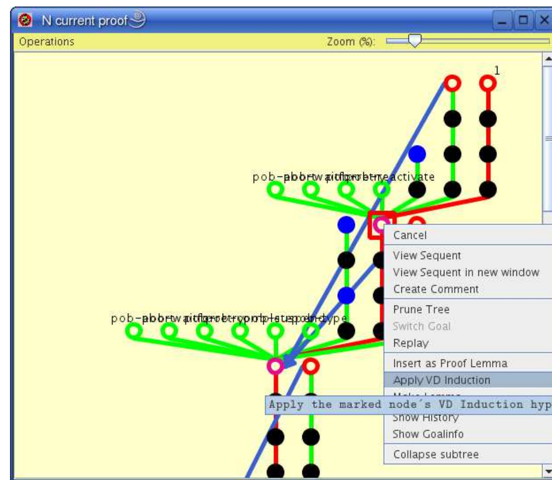

Figure 4: Using *apply VD induction*

where the asterisk denotes, 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 or not 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 again 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 [2] 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 4).

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

However, 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.

In principle it is suggested to use the rule set `TL heuristics`, with the most important rules being the simplifier and the application of module specific rules.

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 PrescribeObservation. Our proof obligation is as follows:

```
  (: system description :)
  [asbru-plan#(‘pob’, ‘ob’; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG)],

  (: current system configuration :)
  AS[‘pob’] = inactive,

  (: environment assumption :)
  \G (    AS’’[‘pob’] = AS’[‘pob’]
      and not suspendedP(AS’’[‘ob’]))

|- (: property :)
```

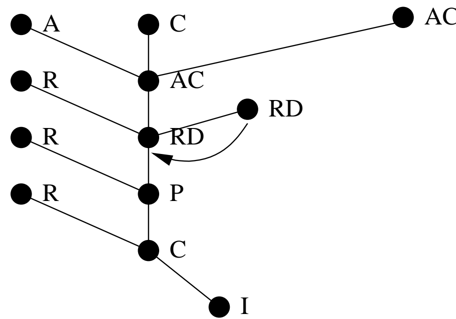

Figure 5: Proof tree for PrescribeObservation

```
\G (    [Frame AS, Tick]
    and frame(AS', AS, 'pob' '));
```

A proof graph can be found in Figure 5.

5.1 Induction

The property is a safety property. Therefore, it is possible to use the property for induction. Use rule VD induction and select the safety property for induction.

5.2 First step

In the initial configuration, plan `'pob'` is inactive. After the first step has been executed with rule step, plan `'pob'` is considered.

In principle, there is a second goal, in which the plan `'pob'` is still inactive, because the parent did not send the consider signal. In such cases, the representing program is then terminated. As the frame properties are both satisfied, if the program terminates, this goal can be automatically closed.

5.3 Further steps

In every further step, the possibility of plan `'pob'` terminating has to be considered. Once a plan terminates, its representing program will terminate and therefore, with the same reason as given above, the side arm of the proof can be closed.

Execution continues until plan `'pob'` is ready. The system configuration now reads

```
AS['pob'] = ready
```

After executing the next step, we receive two premises. In the first premise, the activate signal has been sent and `'pob'` is activated

```
AS['pob'] = activated([], [], [])
```

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

```
AS[‘pob’] = ready
```

This premise can be solved with induction.

Proof continues with an activated `‘pob’` plan. After the execution of another step, we again receive two goals. In the first two premises the plan is terminated. This goal can be closed easily.

In the third premise, `‘pob’` is still activated. We can solve this goal again with induction. Therefore this proof is completed.

Note the assumption, that `‘ob’` will never be in state suspended. This assumption is not strictly necessary, that is, the property could be verified even without that knowledge. However, the proof would be considerable larger. As in the complete jaundice case study there is no suspend condition, plans can suspend iff there parents suspend. As there is no suspending parent, there will never be a suspended plan in Jaundice.

6 Verifying a small hierarchy of Asbru plans

We try to verify a property of Observation. This plan refers to a sub plan Prescribe-Observation. Our proof obligation is as follows:

```
ob-intention:
  (: system description :)
  [asbru-plan#(‘ob’, ‘rt’; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG)],

  (: current system configuration :)
  AS[‘ob’] = inactive,
  Tick,

  (: environment assumption :)
  \G (    AS’’[‘ob’]  = AS’[‘ob’]
      and AS’’[‘pob’] = AS’[‘pob’]
      and PC’’[‘pob’] = PC’[‘pob’]
      and not suspendedP(AS[‘rt’])
      and Tick’’)
|- (: property :)
  \G (  AS’[‘ob’] = completed and AS[‘ob’] \neq completed
      -> PDH[AC][‘bilirubin’] .val = observation );
```

A proof graph can be found in Figure 6.

6.1 Induction

The property is a safety property. Therefore, it is possible to use the property for induction. Use rule VD induction and select the safety property for induction.

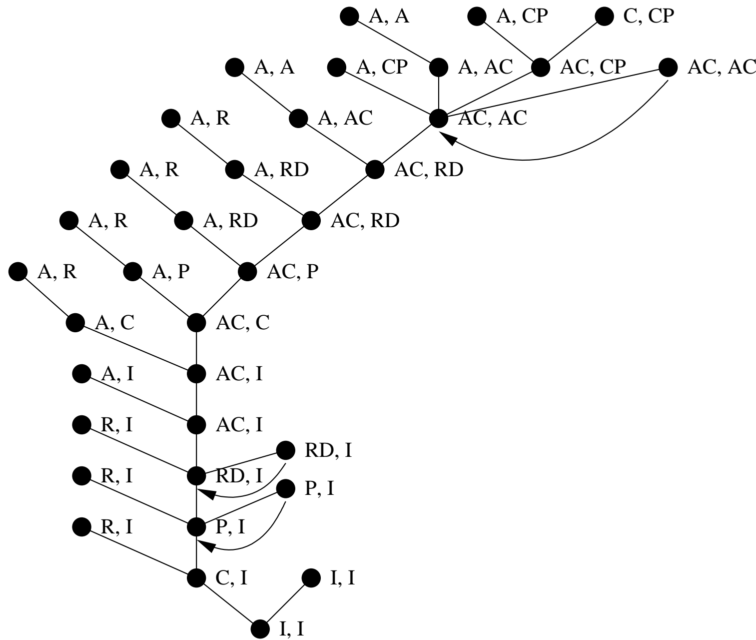

Figure 6: Prooftree of Observation

6.2 First step

In the initial configuration, plan `‘ob’` is inactive. After the first step has been executed with rule step, plan `‘ob’` is considered.

As with the proof of the single plans, there is a second premise in which no consider signal was sent to `‘ob’`. In this additional premise, the execution of the system immediately terminates with `‘ob’` being inactive. As the property holds trivially in this case, that goal can be closed automatically.

6.3 Further steps

Execution continues as described in Section 5.3 until plan `‘ob’` is ready. When `‘ob’` gets activated, it will set the status of all children to state inactive. After activation, the system configuration reads

```
AS[‘ob’] = activated([], [], ‘pob’ ’, 1),
AS[‘pob’] = inactive
```

Plan `‘ob’` is activated and its sub plan is listed in the third list, as it is yet inactive and `‘ob’` has not yet sent any signals. Plan `‘pob’` is now known to be inactive.

After executing the next step, we receive the configuration

```
AS[‘ob’] = activated([], ‘pob’ ’, [], 1),
AS[‘pob’] = inactive
```

In addition, the consider signal is sent to the sub plan:

```
PC['pob'] = mk-pce(true, false, false)
```

Thus, the sub plan will change its state to considered in the next step and the resulting configuration reads

```
AS['ob'] = activated([], 'pob' ', [], 1),
AS['pob'] = considered
```

6.4 Activating the sub plan

Proof continues with the execution of `'pob'`. The sub plan passes through configurations considered, possible and ready. In every step, we receive additional premises: it is possible that `'ob'` is aborted. This can be, because the abort condition of `'ob'` was satisfied, the parent was aborted or the parent was completed. In all of these cases, it takes another step for the state of `'ob'` to propagate to `'pob'`.

After `'pob'` is ready, Observation sends signal activate and copies the sub plan to its list of activated plan. The configuration reads:

```
AS['ob'] = activated('pob' ', [], [], 1),
AS['pob'] = ready
```

Therefore, although there are two states in direct sequence with the same Asbru configuration, it is neither possible nor necessary to use induction. In the next state `'pob'` will be activated.

6.5 Environment interaction

Immediately after the sub plan `'pob'` has been activated, the patient data is as follows:

```
(PDH[AC]['bilirubin']) = mk-pd(observation)
```

i.e., the current level of bilirubin is observation. The system waits for an environment change. Thus, induction is necessary and the patient data must be generalised. As the environment is not restricted, bilirubin can be any level in the next step. Therefore, the formula above is discarded.

Executing another step gives four premises:

- Observation is aborted, because bilirubin is too high.
- The parent of Observation terminates. Therefore `'ob'` will abort itself.
- Bilirubin is still of level Observation and Prescribe-Observation completes.
- Bilirubin is still of level Observation and Prescribe-Observation remains activated.

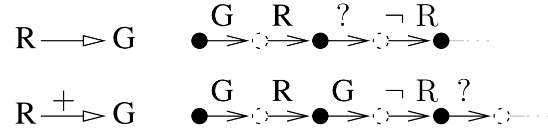

Figure 7: Visualisation of Rely Guarantee

The first two premises are trivial. In the third premise, it takes another step for Observation to complete. The fourth premise can be solved with induction.

As with the verification of prescribe observation, it is again safe to assume, regular treatments will not suspend, as there are no suspend conditions in Jaundice. The proof size is reduced from around 160 to 43 interactive steps.

7 Compositional verification of large hierarchies

7.1 Rely guarantee in theory

For larger plan hierarchies the proof method as described in section 6 is not feasible. That is, with the number of subplans in indeterministic or parallel controls, the complexity of the proof grows exponentially. Therefore it is necessary to use a different approach towards verification of such hierarchies.

Our suggestion is the incorporation of the rely guarantee approach $\stackrel{+}{\rightarrow}$ as proposed by Lamport. In short, the focus is shifted away form a global observer towards the point of view of the individual component. Instead of an explicit modelling of the environment with all the restrictions in its behaviour, the component specifies the subset of the environmental behaviour necessary for itself to function properly. This is called the rely of the component.

If the rely of the component is satisfied, that is, the environment behaves accordingly, the component guarantees a certain behaviour itself. This behaviour is called guarantee. Should the environment violate the rely, the component is also allowed to violate its guarantee. The time span between violation of the rely and violation of the guarantee can be specified by using either the $R \stackrel{+}{\rightarrow} G$ or the $R \rightarrow G$ operator. The latter states, that the guarantee must not be violated as long as the rely is not violated, the former even states that the guarantee has to be valid within the step, where the rely is violated. The interpretation is, that the component cannot have noticed the violation and is therefore obliged to establish the guarantee at least until this step.

Figure 7 is a visualisation of $\stackrel{+}{\rightarrow}$ and $\rightarrow$, where the G/? marked arrows represent system transitions and the R/¬ R marked arrows represent environment transitions. The ? marked arrows denote the first system transition, where the system may behave arbitrary. In the visualisation it can clearly be seen, that $\rightarrow$ allows for visionary systems. The system may violate the guarantee even before it has the chance to even note this.

Assuming a system, where the rely cannot hold, once the Guarantee is violated,

one can easily see, that this is not a wanted behaviour.

Within KIV the rely guarantee operator $R \overset{+}{\rightarrow} G$ is translated to `G unless (G and not R)`.

7.2 Verification of basic rely guarantee properties

The basic case for rely guarantee is to verify the validity of a rely guarantee property by means as described in section 6. However, for the property to be usable as a RG property it has to be of a syntactical different form. While basic properties adhere to the form

```
configuration,
system,
environment assumption |-
property
```

RG sequents have to adhere to this form

```
   system
->    configuration,
   ->        property (guarantee)
      unless     property (guarantee)
             and not environment assumption (rely)
```

This form is necessary, so that it is possible to rewrite the program in a later state. If at some point the system is running in parallel to another system description this can be rewritten, like

```
       configuration,
   ->        guarantee
      unless guarantee and not rely
||s system2
-----------------------------------
       system ||s system2
```

After this step, it should be possible to establish the validity of the configuration via simplification. After this has been done, the sequent looks like

```
(guarantee unless guarantee and not rely) ||s system2
```

This sequent has to be established by further proof decomposition or by plain symbolic execution.

```
|-
  (: system :)
  [asbru-plan#('ob', 'rt'; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG)]
-> (: configuration :)
```

```
(      AS[‘ob’] = inactive or AS[‘ob’] = considered
    or AS[‘ob’] = possible or AS[‘ob’] = ready
 -> (: guarantee :)
    (  (  (AS’)[‘ob’] = (completed) and not AS[‘ob’] = (completed)
     ->   (PDH[AC][‘bilirubin’]) .pdvalue .val = (observation)
       and \W laststep)
    and frame(AS, AS’, ‘ob’ + ‘pob’ ’)
    and frame(PC, PC’, ‘pob’ ’)
    and [FRM AS, PC, Tick]
  unless (: guarantee :)
         (  (  (AS’)[‘ob’] = (completed) and not AS[‘ob’] = (completed)
             ->       (PDH[AC][‘bilirubin’]) .pdvalue .val
                    = (observation)
                and \W laststep)
             and frame(AS, AS’, ‘ob’ + ‘pob’ ’)
             and frame(PC, PC’, ‘pob’ ’)
             and [FRM AS, PC, Tick]
             and not (: rely :)
                    (    AS’’[‘pob’] = AS’[‘pob’]
                     and AS’’[‘ob’]  = AS’[‘ob’]
                     and PC’’[‘pob’] = PC’[‘pob’]))))
```

Verification of this property closely resembles the verification described in 6, however, the different form of the property results in more cases after each step. These additional goals are a result of incomplete simplifications, where predicate logic contradictions could not be automatically determined. They can be closed by the predicate logic simplifier after application of lemmas.

The similarity within the proof is no coincidence. When looking closely at the RG sequent, one will notice that the essential property is exactly the same as the one in the previous section, that is, the property to prove is that if observation completes, the bilirubin level is observation.

7.3 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 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 singular 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 unless formulas can generate more indeterminism than one singular RG property.

The suggested procedure is to determine a RG property which is a generalisation of singular 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. There

has not yet been found a general rule for the construction of the resulting RG property, although there exist such rules for special cases of plans.

There has been found a compositional theorem to reduce the combination of two RG properties into one to a mere predicate logic one. However, currently there is no automated support for the application of this theorem. Therefore, application of this theorem can only be done manually outside the KIV system.

In many cases the resulting rely is merely a combination of both singular relies and the same holds for the guarantees. For example in one proof fusing two RG properties for the asbru plans 'prescribe observation manual activation' and 'prescribe normal phototherapy manual activation' the relies for both singular properties were

```
(AS'')[‘pobma’] = (AS’)[‘pobma’]
```

and

```
(AS'')[‘pnpma’] = (AS’)[‘pnpma’]
```

the resulting rely is

```
    (AS'')[‘pobma’] = (AS’)[‘pobma’]
and (AS'')[‘pnpma’] = (AS’)[‘pnpma’]
```

which is exactly the conjunction of both RG relies.

A similar combination can be found with the combination of the guarantees. These are the two singular guarantees:

```
    [FRM Tick, AS]
and frame(AS’, AS, ‘pobma’)
```

and

```
    [FRM AS, Tick]
and frame(AS’, AS, ‘pnpma’)
```

The combination is again something similar as the conjunction of both properties, however, the use of the predicate frame makes things a little bit more complex here, as `frame(AS’, AS, ‘pnpma’)` $\wedge$ `frame(AS’, AS, ‘pobma’)` is equivalent to `frame(AS’, AS, [])`. The correct combination here is therefore

```
    [FRM AS, Tick]
and frame(AS’, AS, ‘pnpma’ ’ + ‘pobma’ ’)
```

As the proof obligation consists of unless formulas, it is necessary to use induction for the proof. However, as there has not yet been gathered much experience with the combination of multiple RG properties, details of the proof will not

be described here until it can be established, they are in principle valid beyond this property.

It should be noted, that the properties above are typically in the sense, that completion or abortion of the user performed children does not influence the property, for which the abstraction should be used in this example. In such cases, usually the RG properties are similar to the above mentioned.

In such cases, where user performed children do not influence the verification of the property by their termination behaviour, it is also feasible to verify, that the guarantee always holds, even with no rely present. Therefore, a generic property has been defined with which all user performed children can be rewritten. The property is:

```
  [asbru-plan#(sk, sk1; Tick, Patient, PDH, VH, AC, ASH, AS, EAGG)]
-> asbru(sk) .type = user
   -> \G [FRM AS, Tick] and frame(AS’, AS, sk ’)
```

It has been defined within the Asbru specification and can therefore be used in all case studies without adaption or re verification.

We think, that there is great potential with those generic properties and that they might be the key for more efficient interactive verification. We still have to evaluate what the limits of this technique is.

7.4 Using rely guarantee properties

The properties described in the section 7.2 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 paragraph, where at least two hierarchy levels with plans observation and prescribe observation are both reduced to the much simpler unless formula. It is not hard to see, this technique can also be applied to more complex plans like regular treatments.

This section will describe how to use the abstracted properties with the verification of a RG property formulating a statement regarding the plan phototherapy-recommendation. This plan is applied when the level of bilirubin is slightly elevated. With this level of bilirubin, it is left to the doctor to decide, whether to observe the bilirubin level and wait for a decrease to level observation or, alternatively, prescribe phototherapy to accelerate the reduction of bilirubin.

Therefore, phototherapy recommendation has two subplans, ’prescribe observation’ and ’prescribe phototherapy normal’, that are executed with an any order control.

As it is not required for phototherapy recommendation to yield any result, the RG property formulated only postulates some sanity conditions. Those are, that in every step of the phototherapy recommendation plan execution, asbru state fields that are not under the direct influence of ptr or its subplans are left alone. Furthermore, all other variables are not changed by ptr.

To verify this property, two RG properties for the subplans of ptr have been formulated. those state similar properties as the property formulated for ptr but scaled down to the respective plans.

It has been established, that the verification of two parallel RG properties in addition to the super plan is a very complex matter. Therefore it is suggested to combine the RG properties in a different proof (for details see section 7.3).

The initial property is similar to the one described in 7.2. For reasons of completeness, it is written down here and shortly explained.

```
|-
  (:- system -:)
  [asbru-plan#(‘ptr’, ‘rt’; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG)]
-> (:- configuration -:)
   (      (   AS[‘ptr’] = (inactive) or AS[‘ptr’] = (possible)
          or AS[‘ptr’] = (considered) or AS[‘ptr’] = (ready)
     and Tick
   -> (:- guarantee -:)
      (    [FRM AS, PC, Tick]
       and frame(AS’, AS, ‘ptr’ ’ + ‘pobma’ ’ + ‘pnpma’ ’)
       and frame(PC’, PC, ‘pobma’ ’ + ‘pnpma’ ’))
     unless (:- guarantee -:)
            (   laststep
             or     [FRM AS, PC, Tick]
                and frame(AS’, AS, ‘ptr’ ’ + ‘pobma’ ’ + ‘pnpma’ ’)
                and frame(PC’, PC, ‘pobma’ ’ + ‘pnpma’ ’)
                and not (:- rely -:)
                        (    AS’’[‘ptr’]   = AS’[‘ptr’]
                         and AS’’[‘pobma’] = AS’[‘pobma’]
                         and AS’’[‘pnpma’] = AS’[‘pnpma’]
                         and PC’’[‘pobma’] = PC’[‘pobma’]
                         and PC’’[‘pnpma’] = PC’[‘pnpma’])));
```

As long as the environment guarantees not to change the signals sent by ptr to its children and as long as the Asbru state is left alone, the plan guarantees not to touch anything but the Asbru state fields of itself and its children and no other signals may be sent, than those to the children of the plan ptr.

The property is symbolically executed in the manner described in section 6 to the point, where the subplans of ptr are spawned. After the spawning, the subplans are rewritten by their respective RG properties.

```
  [ begin
      asbru-plan#(‘pobma’; Tick, Patient, PDH, VH, AC, ASH, AS, EAGG) ||s
      asbru-plan#(‘pnpma’; Tick, Patient, PDH, VH, AC, ASH, AS, EAGG)
    end ],
  AS[‘pobma’] = inactive,
  AS[‘pnpma’] = inactive


    begin
```

```
    [PL    AS[‘pobma’] = inactive
        ->     [FRM AS, Tick]
           and frame(AS’, AS, ‘pobma’)
        unless (    [FRM AS, Tick]
                and frame(AS’, AS, ‘pobma’ ’)
                and not (AS’’)[‘pobma’] = (AS’)[‘pobma’])
    ] ||s
    [PL    AS[‘pnpma’] = inactive
        ->     [FRM AS, Tick]
           and frame(AS’, AS, ‘pnpma’)
        unless (    [FRM AS, Tick]
                and frame(AS’, AS, ‘pnpma’ ’)
                and not (AS’’)[‘pnpma’] = (AS’)[‘pnpma’])
    ]
end
```

After this rewrite step, the properties are linked together by the lemma described in section 7.3. The outcome of this lemma application is

```
[PL (    [FRM AS, Tick]
     and (AS’, AS, ‘pnpma’ ’ + ‘pobma’ ’))
    unless (    [FRM AS, Tick]
            and (AS’, AS, ‘pnpma’ ’ + ‘pobma’ ’)
            and not ((AS’’)[‘pobma’] = (AS’)[‘pobma’]
                     (AS’’)[‘pnpma’] = (AS’)[‘pnpma’]))
]
```

After this lemma application, the knowledge about the states of pnpma and pobma is thrown away. Experience shows, that this step, although it seems drastically is instead greatly reducing the complexity of the proof. As the RG property no longer contains information about the progression of the Asbru states, the knowledge about the current states is no longer important. After the first step the abstracted subplans may have reached any arbitrary asbru state regardless the knowledge they were inactive before. However it is not possible to close some of the open goals by induction, as the plans have reached different states but inactive.

This drastic procedure is only advisable, as long as the property for the subplan does not contribute in an active sense, i.e. as long it is only a sanity condition. 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 properties and formulation of them is a creative step.

If the configuration of the abstracted plan is weakened away, almost all of the newly generated goals can be closed almost instantly by contradictions in the configuration or by induction. Only in few goals, where either the children or the parent plan were aborted and their program terminated, no induction is possible. Instead, it is necessary to further step there.

This is the goal before application of the step rule

```
[begin
```

```
    asbru-plan#(‘ptr’; Tick, Patient, PDH, VH, AC, ASH, AS, EAGG) ||s
    [PL (    [FRM AS, Tick]
         and frame(AS’, AS, ‘pnpma’ ’ + ‘pobma’ ’))
        unless (   laststep
                or     [FRM AS, Tick]
                   and frame(AS’, AS, ‘pnpma’ ’ + ‘pobma’ ’)
                   and not ((AS’’)[‘pobma’] = (AS’)[‘pobma’]
                            (AS’’)[‘pnpma’] = (AS’)[‘pnpma’]))
    ]
 end
],
N = N’’ + 1 until Boolvar,
INDHYP,
Tick,
AS[‘ptr’] = activated(skx0, sky0, []),
all sk. sk \in skx0 or sk \in sky0 -> sk = ‘pobma’ or sk = ‘pnpma’,
|-
(    [FRM AS, PC, Tick]
 and frame(AS’, AS, ‘ptr’ ’ + ‘pnpma’ ’ + ‘pobma’ ’)
 and frame(PC’, PC, ‘pnpma’ ’ + ‘pobma’ ’))
unless (   laststep
        or     [FRM AS, PC, Tick]
           and frame(AS’, AS, ‘ptr’ ’ + ‘pnpma’ ’ + ‘pobma’ ’)
           and frame(PC’, PC, ‘pnpma’ ’ + ‘pobma’ ’)
           and not (    (AS’’)[‘pobma’] = (AS’)[‘pobma’]
                    and (AS’’)[‘pnpma’] = (AS’)[‘pnpma’]
                    and (AS’’)[‘ptr’]   = (AS’)[‘ptr’]
                    and (PC’’)[‘pnpma’] = (PC’)[‘pnpma’]
                    and (PC’’)[‘pobma’] = (PC’)[‘pobma’])
        or    [FRM AS, PC, Tick]
          and frame(AS’, AS, ‘ptr’ ’ + ‘pnpma’ ’ + ‘pobma’ ’)
          and frame(PC’, PC, ‘pnpma’ ’ + ‘pobma’ ’)
          and Boolvar)
```

Looking closely, one will notice the plan lists describing the running and evaluated subplans of ptr 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 abstractions lead 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.

The next depicted sequents are somewhat typical representatives of different classes of generated goals.

```
N = N’’ + 1 until Boolvar,
[begin
    asbru-plan#(‘ptr’; Tick, Patient, PDH, VH, AC, ASH, AS, EAGG) ||s
    [PL (    [FRM AS, Tick]
         and frame(AS’, AS, ‘pnpma’ ’ + ‘pobma’ ’))
        unless (   laststep
```

```
                or      [FRM AS, Tick]
                   and frame(AS', AS, 'pnpma' ' + 'pobma' ')
                   and not ((AS'')['pobma'] = (AS')['pobma']
                            (AS'')['pnpma'] = (AS')['pnpma']))
    ]
 end
],
INDHYP,
as['ptr'] = activated(skx0, sky0, []),
PDH[AC] = pdh[ac],
Tick,
AS['pnpma'] = as1['pnpma'],
AS['pobma'] = as1['pobma'],
AS['ptr'] = activated([], [], []),
(pdh[ac]['bilirubin']) = mk-pd(phototherapy-recommendation),
frame(as1, as, 'pobma' ' + 'pnpma' '),
sync
(as, as['ptr', activated([], [], [])], as1,
 as1['ptr', activated([], [], [])]),
not abortedP(ash[ac]['rt']),
all sk. sk in skx0 or sk in sky0 -> sk = 'pobma' or sk = 'pnpma',
all sk. sk = sk0 or sk in skx1 or sk in sky -> sk = 'pobma' or sk = 'pnpma',
not (  (abortedP(as['pobma']) or rejectedP(as['pobma']))
   and (  not abortedP(as['pobma']) and not rejectedP(as['pobma'])
      or abortedP(as['pnpma']) or rejectedP(as['pnpma'])))
|- (: unless omitted for better readability :)
```

This is one of several goals, that can be closed by induction. For reasons of improved readability, the unless formula in the succedent has been omitted. Furthermore, it has been chosen to present the simplified sequent. Although much knowledge has been generated, regarding for example the state of the abstracted subplans, most of this is unnecessary to apply the VD induction. The crucial parts of this sequent are the unchanged program as well as the following PL formula

```
AS['ptr'] = activated([], [], []),
```

This formula states, that the state of ptr has not changed during the step. The lists of running and evaluated subplans have been made explicit. That is, instead of the skx and sky written before the step, there are now written down empty lists. Not even the all quantified pl formulas are needed here. Originally they restricted the elements within the skx and sky lists. By definition no element is in the empty list, therefore, the formulas can be discarded, as it can be concluded that not more then pobma and pnpma are in the empty lists.

Another typical example for a generated goal is this:

```
[asbru-plan#('ptr', 'rt'; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG)],
N = N'' + 1 until Boolvar,
INDHYP,
```

```
AS[‘ptr’] = activated([], [], [])
|- (: unless omitted for better readability :)
```

In this goal, the RG unless representing the subplans has terminated. Therefore only ptr itself is still running. Because of the termination of the children, represented by the antecedent unless formula, it is no longer possible to apply the induction to the underlying goal. Instead, it is necessary to apply further steps. After a further step there are two possibilities, first, ptr itself is terminated, in which case the proof can be closed automatically or ptr is still active. In that case, a similar goal

```
[asbru-plan#(‘ptr’, ‘rt’; Tick, Patient, PDH, VH, AC, ASH, AS, PC, EAGG)],
N = N’’ + 1 until Boolvar,
INDHYP,
(pdh[ac][‘bilirubin’]) = mk-pd(phototherapy-recommendation),
AS[‘ptr’] = activated([], [], []),
AS[‘pobma’] = as[‘pobma’],
AS[‘pnpma’] = as[‘pnpma’],
Tick,
not (as[‘rt’]) = (completed),
not (as[‘pobma’]) = (completed),
not (as[‘pnpma’]) = (completed),
not abortedP(as[‘rt’]),
tick or AC = ac +m 1,
tick or PDH[AC] = pdh[ac],
not (  (abortedP(as[‘pobma’]) or rejectedP(as[‘pobma’]))
   and (  not abortedP(as[‘pobma’]) and not rejectedP(as[‘pobma’])
      or abortedP(as[‘pnpma’]) or rejectedP(as[‘pnpma’])))
|- (: unless omitted for better readability :)
```

is generated. Abstracting from some of the unimportant formulas there, one can notice that the important PL formulas for the application of induction are still there, that is, ptr is still active.

It can, in some cases, be necessary to abstract from the lists of the activated state 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. Therefore if the current state is, for example,

```
AS[‘ptr’] = activated([], ‘pobma’ ’, [])
```

after a further step, this state could be rewritten to

```
AS[‘ptr’] = activated(‘pobma’ ’, [], [])
```

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.

7.5 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 always the same 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 automatise with the necessity of instantiation of quantifiers or the application of lemmas that - on first sight - can not be applied by the simplifier.

References

[1] 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.

[2] 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.

[3] J. Schmitt, M. Balser, and W. Reif. Support for interactive verification of Asbru in KIV. Technical report, University of Augsburg, 2005. To appear.