



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 deliverable
D6 Living Proofs**

Due date of deliverable: 30 September 2005

Actual submission date: 30 September 2005

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

Organisation name of lead contractor: Universitat Jaume I

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 D6: Living proofs (prototype)

Michael Balser, Jonathan Schmitt, Wolfgang Reif
University of Augsburg

September 29, 2005

Abstract

Currently, medical guidelines are revised every 5-7 years. The goal of this work-package is to provide a method for developing good quality medical guidelines and to efficiently revise these guidelines every 2 years without compromising the established quality. Three strategies are proposed: (I) construction of counter examples to help locating errors, (II) automatic analysis to determine affected properties after guideline has been changed, and (III) replay of proofs to assist in construction of new proofs for modified guideline. The three strategies have been prototypically integrated into KIV, an interactive verification environment. The strategies are rounded of with tool support to track changes between informal guideline and formalization as it is implemented in the Markup & Editing tool DELTA (see Deliverable D2.3a).

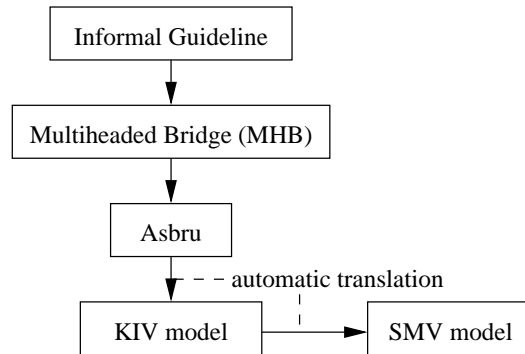


Figure 1: Levels of formalization

Contents

1 Introduction	4
2 Counter Examples	5
2.1 Counter examples in KIV	5
2.2 Rule <code>show asbru config</code>	7
2.3 Rule <code>show asbru trace</code>	9
2.4 Limitations	9
3 Correctness Management	9
3.1 Introduction	9
3.2 Integration of Asbru	10
3.3 Methodology for Asbru	11
4 Replay of proofs	13
5 Prototype	15
6 Conclusion	15

1 Introduction

Currently, medical guidelines are revised every 5-7 years. The goal of this paper is to provide a method for developing good quality medical guidelines and to efficiently revise these guidelines every 2 years without compromising the established quality.

In our research project Protocolure, we assure the quality of medical guidelines by formalizing the informal guideline and by verifying the desired properties (e.g. indicators). Formalization follows different levels which are depicted in Figure 1. Constructing a formal model alone, i.e. translating it to a formal language, enhances the quality of the original guideline. Furthermore, verification of the property by theorem proving or automatic model checking uncovers anomalies in guidelines. If a property cannot be verified, we construct counter examples to

efficiently locate the error source. As a result, we receive guidelines of highest quality.

If the informal guideline changes, the formal model must be adapted and the properties must be re-verified. Our goal is to reduce the effort of quality assurance, if a medical guideline is changed.

Our approach makes use of a tool named DELTA which administrates links between the different levels of formalization. These links can be exploited to track changes; if part of the informal guideline is changed, the corresponding parts in the formal model must also be changed. Formal verification of properties in the theorem prover KIV gives dependencies between parts of the formal model and properties. If the model is changed, an automatic analysis determines those properties which are affected ("correctness management"). Affected proofs become invalid. However, they can be partially replayed to efficiently construct a new proof.

This deliverable reports on three techniques to facilitate the validation and verification of evolving guidelines:

- counter examples (see Sect. 2)
- correctness management (see Sect. 3)
- replay of proofs (see Sect. 4)

For managing living guidelines, we propose to combine the three techniques of this work-package with the tool DELTA which can be used to administrate links between the different levels of formalization (see Deliverable D2.3a).

2 Counter Examples

We are interested in the verification of properties for medical guidelines. For the verification of properties, we have integrated Asbru into the interactive theorem prover KIV. Furthermore, we have defined a calculus for the symbolic execution of guidelines. With symbolic execution and induction, intuitive proofs can be automatically constructed to a large extent (see Deliverable D4.2b). An alternative to interactive verification is model checking (see Deliverable D4.3). In practice, however, proofs often fail because of errors. Either

- the guideline does not satisfy the property under verification, and the guideline or the property must be corrected, or
- the current proof branch leads into a dead-end, and lemmas/invariants (for interactive verification) or abstractions (for model checking) must be corrected.

The source of error can be determined by examination of the proof in the interactive verifier, or examination of counter example in a model checker.

2.1 Counter examples in KIV

In a model checker, counter examples are automatically generated. In KIV, our interactive proof strategy of symbolic execution makes it easy to derive

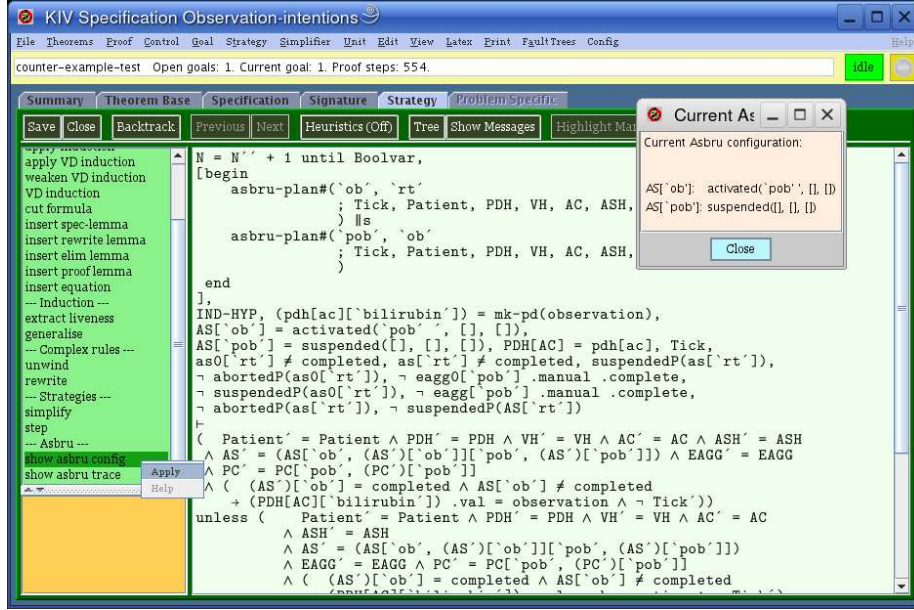


Figure 3: Example Asbru configuration

2.2 Rule show asbru config

The KIV rule `show asbru config` analyses a single node to extract information about the current state of Asbru plans. A screen-shot illustrating the application of this rule can be found in Figure 3. The figure displays the strategy panel of KIV where proofs are constructed. The current premise is displayed in the large area on the lower right. The applicable proof rules are on the left. If rule `show asbru config` is applied, the extracted information is displayed in an additional window which is displayed in the upper right: currently plan 'ob' is activated and 'pob' is suspended.

The information is extracted as follows. The current premise is a sequent

$$\Gamma \vdash \phi,$$

the antecedent containing a number of preconditions Γ , the succedent the temporal formula under verification ϕ . The description of the current state can be found in the antecedent. The preconditions of the example in Figure 3 can be classified as follows: (I) a formula

```

..., [ asbru-plan#('ob', 'rt'; ..., AS, ...)
      ||s asbru-plan#('pob', 'ob'; ..., AS, ...) ], ...
  
```

with the Asbru plans 'ob' and 'pob' running in parallel, (II) temporal formulas containing the induction hypothesis

```

..., N = N'' + 1 until Boolvar, IND-HYP, ... ,
  
```

(III) additional temporal formulas containing assumptions necessary to verify the temporal property (not contained in the example), and (IV) formulas describing the current state

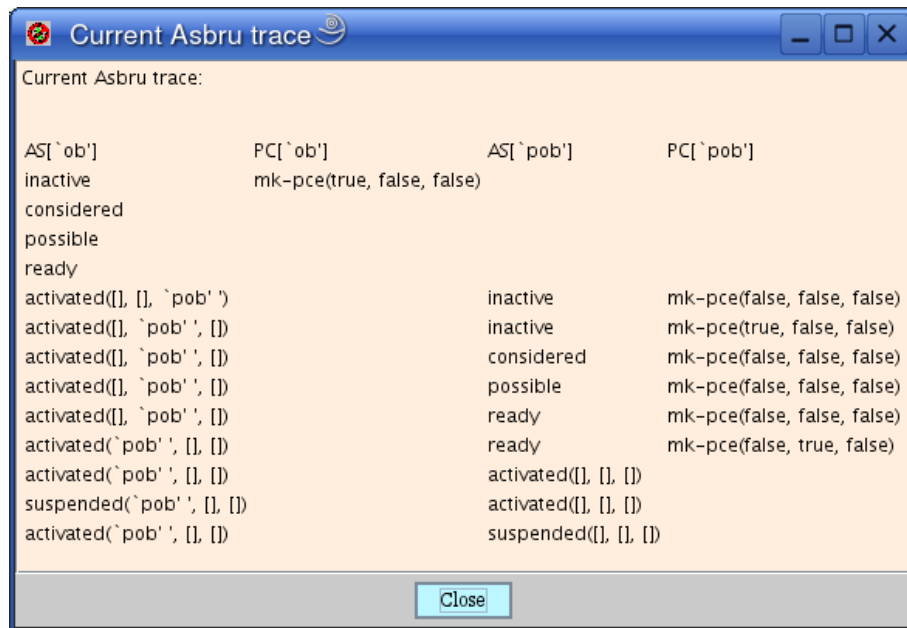


Figure 4: Example Asbru trace

```

...
pdh[ac]['bilirubin'] = mk-pd(observation),
AS['ob'] = activated(`pob` , [], []),
AS['pob'] = suspended([], [], []), PDH[AC] = pdh[ac], Tick,
as0['rt'] /= completed, as['rt'] /= completed,
suspendedP(as['rt']), not abortedP(as0['rt']),
not eagg0['pob'].manual.complete, not suspendedP(as0['rt']),
not eagg['pob'].manual.complete, not abortedP(as['rt']),
not suspendedP(AS['rt'])

```

Formula (I) refers to a dynamic variable AS representing the current Asbru state. Formulas (IV) contain equations

$$AS[\langle \text{plan name} \rangle] = \langle \text{plan state} \rangle$$

referring to the same variable AS and defining the current Asbru configuration. These equations are extracted, and the information is presented for the different plans in alphabetic order. The presentation can be seen in the additional window in the upper right of Figure 3.

Besides information about the current state of Asbru plans contained in variable AS, information about the external Asbru signals contained in variable PC are collected. Every Asbru plan is controlled via three signals consider, activate, and retry. In the example of Figure 3, nothing is known about the values of these signals.

2.3 Rule show asbru trace

The KIV rule `show asbru trace` analyses a complete proof branch and extracts information about the different steps which were executed along this path. Starting from the current premise, the proof tree is traversed back to the root. For each step execution, the current Asbru configuration is extracted as described in the previous section. The list of Asbru configurations is presented in a table as can be seen in Figure 4. For each state, there is a corresponding row in the resulting table with the value for each variable in a separate column. The first step of the proof branch can be found in the first row, the current premise (which is the last step of the proof branch) can be found in the last row. The rows in the example correspond to the annotations of the proof tree in Figure 2.

2.4 Limitations

The extracted information of rule `show asbru trace` represents a good summary of how the current premise has been reached. If the proof for the current premise cannot be completed, this has helped a lot to detect the error in the Asbru plans or the property under verification. The strategy can be extended to include not only information about the current configuration of Asbru plans but also about the current configuration of the patient.

The strategy works best for concrete configurations. However, large proofs must be modularised which involves the abstraction of Asbru configurations. Currently, the counter example only considers equations similar to

```
AS[<plan name>] = <plan state> .
```

In a modular proof, arbitrary predicates can be used to reason about Asbru configurations. In this case, our analysis only returns a partial trace. In practice, the configuration of sub plans is often abstracted, while the configuration of the top-level plan is concrete. Thus, only information about the configuration of the top-level plan is extracted. This partial counter example is still very helpful for locating errors. A complete construction of counter examples for arbitrary first order predicates is not possible.

3 Correctness Management

3.1 Introduction

The so called correctness management of KIV keeps track of properties which were used as lemmas to prove other properties. If a property is changed, then the proofs of all properties which directly refer to the changed property become invalid. Thus, the number of affected proofs is kept to a minimum.

The dependencies between properties result in a so called dependency graph. As an example, see Figure 5. The proof of lemma `sort-02` refers to the lemmas `sort` and `rec`. Lemma `rec` depends on axiom `Ordered`. If the axiom `Ordered` is changed, then the proof of lemma `rec` becomes invalid. The proof of lemma `sort-02`, however, stays valid. Only if lemma `rec` must also be changed, the proof of lemma `sort-02` is affected.

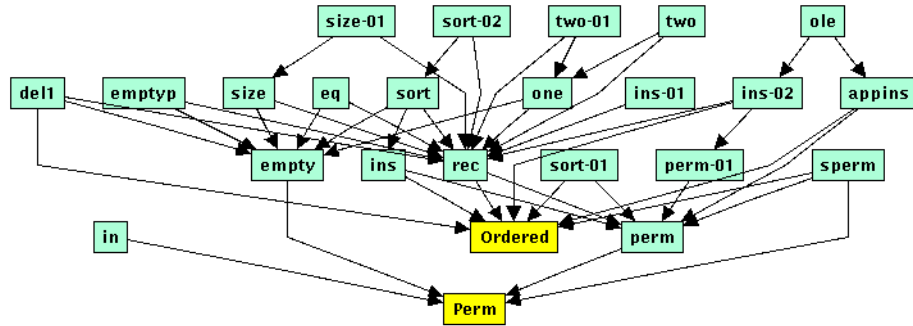


Figure 5: Example dependency graph in KIV

For efficiency, properties are organized in two levels. The first level contains all of the properties for the same specification (local properties), the second level contains all of the properties for sub specifications. If a property changes, proofs in the same specifications will immediately become invalid. Dependency analysis for proofs in other specifications can be invoked with Check Spec Theorems.

3.2 Integration of Asbru

Keeping track of dependencies is very important for the efficient verification of evolving guidelines. Therefore, we have taken special care to support Asbru in KIV such that the correctness management of KIV can be applied. In principle, there are two alternatives to integrate Asbru in KIV:

- Direct support for Asbru in KIV

This is to hard code a data structure for Asbru plans in the programming language of KIV. This approach would require additional effort to incorporate the Asbru formalism in the correctness management. If a proof refers to an Asbru plan definition, the reference must be logged by the correctness management. Changes to plan definitions must invalidate proofs.

- Indirect support for Asbru in KIV

This is to specify Asbru plans in a formalism already implemented in KIV. It is more complex to formally define Asbru plan definitions in algebraic specifications and especially to construct efficient proof support. On the other hand, Asbru incorporates smoothly with the correctness management. Referring to an Asbru plan definition is simply referring to an axiom.

We have decided to go for the second alternative. Additional effort has been spent for the integration of Asbru in KIV. No further effort was necessary to integrate Asbru with the correctness management. Details on the integration of Asbru in KIV can be found in Deliverable D4.2b.

```

axioms

ob-def:
  asbru('ob')
= mk-asbru-def
  (filter_condition,
   mk-aasbruc
   (mk-acond
    (lambda pdh, vh, ash, as, asbru-clock.
      pdh[asbru-clock]['bilirubin'].val = observation),
     false, false),
   suspend_condition,
   resume_condition,
   mk-aasbruc
   (mk-acond
    (lambda pdh, vh, ash, as, asbru-clock.
      (pdh[asbru-clock]['bilirubin']) .val observation),
     false, false),
   complete_condition,
   sequential, false, 'pob' ',
   wait-for-n(1, 'pob' '), false
  );

```

Figure 6: Definition of Asbru plan ob

3.3 Methodology for Asbru

Figure 6 gives an axiom defining Asbru plan `ob` which is the Observation Asbru plans of the Jaundice case study of Protocure I. An Asbru plan definition is a record constructed with function `mk-asbru-def`. The different slots of the record correspond to the different parts of an Asbru plan. The first slot defines the filter condition. The filter condition in the example defaults to value `filter_condition`. The second slot defines the setup condition. In the example, the setup condition requires the bilirubin value of the patient to be equal to `observation`. Setup, resume, abort, and complete condition follow. The body of the plan contains a single sub plan `pob` and is executed sequentially. The plan completes, if and only if `pob` completes (`wait-for-n(1, 'pob')`).

If in a proof, we refer to the Asbru plan `ob`, we refer to the axiom `ob-def`. If the plan definition is changed, then all of the proofs which directly refer to `ob-def` become invalid. In order to achieve a more fine-granular correctness management in practice, we do not directly refer to the plan definition. Instead, we define a set of rules to separately access the different slots of an Asbru plan definition. A selection of these rules can be found in Figure 7. Lemma `ob-filter` can be used to refer to the filter condition, lemma `ob-setup` to refer to the setup condition and so on.

Figure 8 illustrates the usefulness of the additional lemmas. The proof for `ob-intention-term` refers to the filter, setup, and sub-plans slot of plan `ob`. If the abort condition of `ob` is changed, then the proofs for `ob-subplans`, `ob-setup`, and `ob-filter` become invalid. The proof for `ob-intention-term`

```

ob-filter:  asbru('ob').filter = filter_condition;
ob-setup:  asbru('ob').setup = mk-aasbruc(
            mk-acond(
                lambda pdh, vh, ash, as, asbru-clock.
                    pdh[asbru-clock]['bilirubin'].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' ';

:

```

Figure 7: Set of rules to access slots of plan definition

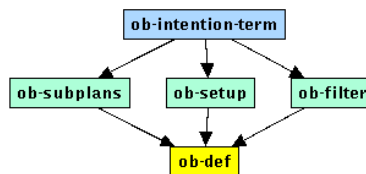


Figure 8: Example dependency graph for Asbru

stays valid. The invalid proofs of the three lemmas can be reconstructed as the respective slots in `ob-def` did not change. Thus, changing the abort condition of plan `ob` did not affect the validity of the proof of `ob-intention-term`.

4 Replay of proofs

The correctness management of KIV ensures that a minimum number of proofs are invalidated. Invalid proofs, however, must be reconstructed. In practice, the new proof is often very similar to the old proof as only small portions of the model or lemma have been changed. It is often sufficient to adapt the different rules in the proof tree to the slightly changed premise. This is what we call *replay of proofs*.

As Asbru has been built on top of an already existing formalism in KIV, it integrates to some extent with the replay mechanisms in KIV. In order to replay a proof, the single steps of the old proof must be adapted to the new premises. Suppose that in the old proof, a rule has been applied on the sequent

$$\phi_1, \dots, \phi_n \vdash \psi_1, \dots, \psi_m .$$

The new sequent reads

$$\phi'_1, \dots, \phi'_{n'} \vdash \psi'_1, \dots, \psi'_{m'}$$

We first try to match the old formulas ϕ_i and ψ_j to the new formulas ϕ'_k and ψ'_l to define a correspondence between old and new formulas. Depending on the rule, a certain degree of inaccuracy is allowed while matching formulas. The resulting correspondence is then used to adjust the parameters of the rule. Finally, the rule is applied with the new parameters to the new sequent.

Finding a correspondence between old and new formulas works the better, the fewer formulas there are. In practice, we try to eliminate irrelevant formulas with simplification. As an example, consider

$$n = 1, n_0 = n + 1 \vdash \phi(n_0)$$

The first precondition $n = 1$ can be eliminated by replacing every occurrence of n with 1, leading to

$$n_0 = 1 + 1 \vdash \phi(n_0)$$

Precondition $n_0 = 1 + 1$ can also be eliminated:

$$\vdash \phi(1 + 1)$$

Thus, the number of formulas is significantly reduced.

This strategy can also be used for replaying Asbru proofs. However, simplifying a premise and eliminating equations did not work as well as expected. As a consequence, rule parameters were not always adjusted as well as possible. A typical Asbru example is displayed in Figure 9. Due to the symbolic execution strategy, equations similar to

```
as['ite-ob'] = ready
```

```

\l
  [asbru-plan#('ite-ob', 'rt'; ...)],
  N = N' + 1 until Boolvar, IND-HYP, Tick, PDH[AC] = pdh[ac],
  PC['ite-qob'] = mk-pce(false, false, false),
  AS['ite-qob'] = inactive,
  PC['ite-pob'] = mk-pce(false, false, false),
  AS['ite-pob'] = inactive,
  AS['ite-ob'] = activated([], [], 'ite-pob' + 'ite-qob'),
  as['ite-ob'] = ready,
  pc['ite-ob'] = mk-pce(boolvar, true, boolvar1),
  not as['rt'] = completed, not suspendedP(AS['rt']),
  not abortedP(as['rt'])
|-
  ...
\l

```

Figure 9: Example Asbru premise

frequently occur with a selector function to access the store `as` on the left and a constant term on the right. These equations could not be eliminated, as the simplifier in KIV only considered equations with a single variable on the left. We have implemented a strategy to also eliminate equations with store access functions.

A strategy to automatically eliminate equations must ensure that the result is still equivalent to the original premise. Under certain conditions, access to a store can be treated like a variable and the equation can be eliminated without weakening the original preconditions. Consider equations

$$v[k] = t$$

with a variable v , a term k which serves as the key and a term t which serves as the assigned value. This variable can be eliminated, if the following requirements are satisfied:

- The variable v represents a store.
- The key k is a constant.
- The store is only accessed with selector functions $v[k']$ with keys k' which are also constants. No other references to the store exists.
- It is decidable, whether the keys k and k' differ.
- Term t does not contain references to $v[k]$, i.e. the equation is not recursive.

These requirements are regularly satisfied in the Asbru context. to eliminate equation $v[k] = t$, we replace every occurrence of $v[k]$ with t and drop the equation afterwards. With this strategy, the premise in Fig. 9 can be simplified to receive the premise in Figure 10.

The described strategy has been prototypically implemented in KIV and has been integrated into the simplification algorithm.

```

[asbru-plan#('ite-ob', 'rt'; ...)],
N = N' + 1 until Boolvar, IND-HYP,
AS['ite-ob'] = activated([], [], 'ite-pob' + 'ite-qob' ),
AS['ite-pob'] = inactive,
PC['ite-pob'] = mk-pce(false, false, false),
AS['ite-qob'] = inactive,
PC['ite-qob'] = mk-pce(false, false, false),
Tick, not as['rt'] = completed, not abortedP(as['rt']),
not suspendedP(AS['rt'])
|-
...

```

Figure 10: Simplified example Asbru premise

5 Prototype

The described techniques have been prototypically implemented in the interactive theorem prover KIV. The KIV system is available on the Internet at

<http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/research>

and is also provided on the CD sent to Brussels as a deliverable. KIV requires

- Linux installation with 'glib' version higher than V2.1
- 96 MB ram (256 MB recommended)
- 1024x768 screen size (or larger) with at least 256 colors
- 350 MB of disk space

KIV has been tested with SuSE and Debian Linux distributions. To start KIV, copy the contents of the CD to your hard disk and start KIV with

```
<kiv installation path>kiv/bin/kiv
```

A tutorial for KIV is provided in the distribution. The Asbru specifications are provided as part of Deliverable D4.2b.

6 Conclusion

Using formal methods in software engineering has shown that an efficient application in practice is possible only, if methods to deal with changes are applied. The same holds for medical guidelines: it takes effort to formalize and verify a medical guideline. If the effort must be repeated over and over again, if the guideline is changed, application of formal methods is rendered useless. Our approach promises to significantly reduce the effort of quality assurance if guidelines are changed.

We have tested our approach with the Jaundice case study of Protocure I. An application to the Breast Cancer case study of Protocure II will be part of Deliverable D4.2c.