Universität Augsburg

Synchronous Parallelism in the Asbru Language

S. Bäumler, M. Balser, W. Reif, J. Schmitt

Report 2008-11 December 2008

Institut für Informatik
D-86135 Augsburg

Abstract

In this paper we present a flexible mechanism for symbolic execution of synchronous parallel programs. The synchronous parallel operator we use allows for techniques like modular reasoning and abstraction of single components. Furthermore, symbolic execution provides intuitive proofs. The operator is included into the interactive higher order theorem prover KIV. We show how to apply our approach using the Asbru medical planning language as an example. This language decomposes medical treatments into many components, which are then executed synchronous parallel.

This work is a joint work that has been partially funded by the DFG program INOPSYS II, under contract number Re 828/6-3 and the European Commission's IST program Protocure II, under contract number IST-FP6-508794.

1 Introduction

Medical guidelines provide the clinical staff with health care recommendations that are based on valid and actual empirical evidence. They are given in form of "systematically developed statements to assists practitioners and patient decisions about appropriate health care for specific circumstances" [1]. In [2] it has been shown that the adherence to guidelines in clinical practice may increase the quality of health care significantly. [3] shows a reduction in treatment costs by using guidelines.

Recent research efforts have applied software verification methods on medical guidelines, in order to increase their quality [4, 5]. These methods help to find ambiguous, incomplete or even inconsistent passages in the natural language guideline documents by stepwise modelling it into a representation with a formal semantics. One of these approaches to apply software verification methods to clinical guidelines was devised in the Protocure project [6]. The underlying idea of this approach is that clinical guidelines can be interpreted as parallel programs. To represent guidelines in a program-like notation we have chosen the language Asbru [7]. So far we have successfully applied the classical verification methods for parallel programs, interactive theorem proving [8] and model checking [9], on Asbru case studies. So far we have dealt with the following case studies: Diabetes mellitus type 2 [10], Jaundice in otherwise healthy newborns [11] and the treatment of breast cancer [12]. All these case studies were initially supplied as natural language texts, then formalized to Asbru models. Formal verification has been applied to these models.

One particular difficulty when dealing with Asbru plans is that typically there are many components which are executed synchronous parallel. For example, in our breast cancer case study we had to deal with up to 49 sub-plans running in parallel and each of these sub-plans increases the complexity of verification.

A common technique for verification of parallel systems is the rely-guarantee

paradigm. It was introduced by Misra & Chandi [13] and by Jones [14]. The underlying idea of this paradigm is that each parallel component guarantees a certain behavior, if its assumptions (the rely) are met by its environment. The key advantage of this technique is, that most of the reasoning is made on single components, not on the complete system and thereby the complexity of the overall proof can be reduced significantly.

In [15] an ITL-based logic with calculus is presented that allows symbolic execution. This calculus was integrated into the interactive theorem prover KIV [16].

Achievements of this Paper are as follows: *I: We present a flexible synchronous parallel operator, which is capable of dealing with clashes and also is compositional. II: We present a modularity theorem which exploits the compositionality of this operator and allows to use the rely-guarantee proof technique. III: We apply this proof technique to the verification of the medical planning language Asbru.*

The paper is structured as follows. We first give an introduction of Asbru, the language we use for describing clinical guidelines in Section 2. In Sect. 3 we give an overview of the temporal logic framework we use while Sect. 4 describes the semantics of the parallel operator. Section 5 describes the modularization theorem we used to apply the rely-guarantee reasoning and we show its application in Section 6. The paper concludes with Sect. 7 where we also discuss related work.

2 Asbru Case Study

Asbru is a hierarchical planning language, especially suited for modeling medical treatments [17]. To model medical treatments Asbru uses the concept of a plans. These plans are organized in a hierarchy where sub-plans represent details of the medical intervention the super-plan describes.

In this paper the breast cancer case study [12] serves as an example. To demonstrate our technique, we take a small excerpt of this case study, concerned with the treatment of a certain stage of the disease called DCIS. Part of the treatment of this disease is the treatment of axilla lymph nodes, which is modeled as a hierarchy of Asbru plans.

The top level plan is named `dwa`. The treatment consists in one of four possible sub-treatments, which are the so called sentinel node procedure, represented by plan `asbSN`, a complete removal of all axilla nodes, represented by plan `cand`, the option to leave the patient untreated, which is represented by plan `gppse` and finally a more sophisticated treatment including not only surgical removal of the lymph nodes but also radiotherapy. This is represented by plan `toan`. This hierarchy is presented in Figure 1.

A super-plan controls the execution of the sub-plans by sending signals. This communication as well as the semantics of the execution have been defined

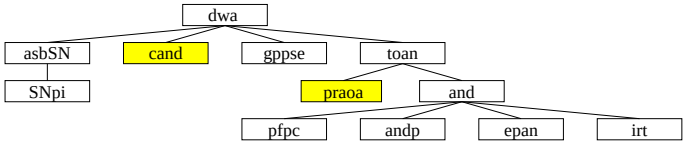


Figure 1: Relevant hierarchy for property 20

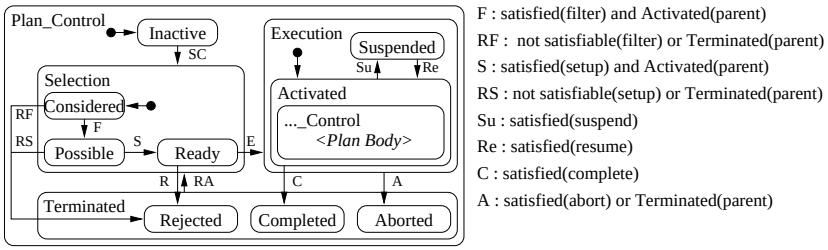


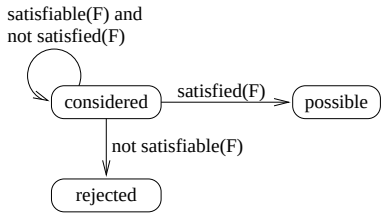
Figure 2: Schematics of parallel execution

in [18]. Core element of this definition is the specification of the dynamic behavior of Asbru using statecharts [19]. The main statechart is depicted in Figure 2.

In this statechart it can be seen that the execution of an Asbru-plan is divided in four different state-groups. Initially the plan is *inactive*. Upon receiving a signal *SC* sent from the super-plan the plan enters the *selection* phase. There the applicability of the plan is determined by conditions named filter- and setup-condition (F and S). These conditions describe circumstances that have to be valid for a plan to proceed in its execution. For example, a plan may only be applicable if the patient is male or has had high blood pressure for at least four hours. If the conditions are satisfied, the guards F and S become true and the plan advances to state *ready*. If either of the conditions is not satisfiable, the plan changes its state to *rejected* via the *RF* or *RS* transitions. In state *ready* a plan expects the signal E sent by its super plan. If this signal is received, control advances to the execution phase, where the sub-plans of the plan are started. Execution in the execution phase is also controlled by conditions, which control the guards *Su*, *Re*, C and A.

2.1 Verification of Asbru

With the specification of the semantics of the Asbru plan state transition model, the most important step towards formal verification of Asbru has been taken. For tool supported verification, it is necessary to embed Asbru into a specification language understood by a verification tool. We opted to embed Asbru in



(a) Asbru semantics

```
considered#(plan; var AS)
begin
  if plan-not-filter(plan, AS) then
    AS[plan] := rejected
    else if plan-filter(plan, AS) then
      AS[plan] := possible;
      possible#(plan; AS)
    else
      AS[plan] := considered;
      considered#(plan; AS)
end
```

(b) KIV implementation

Figure 3: Dynamic behavior of Asbru

the interactive theorem prover KIV using parallel programs.

The statechart semantics definition serves as a foundation for this implementation, and it is possible to keep the implementation very close to these state charts. An example for this is Figure 3(b), where the implementation of part of the semantics shown in Fig. 3(a) is given. The variable AS is of type *asbru-store*. *asbru-store* variables save the plan-states of all plans in one dynamic function. Hence the term $AS[plan]$ selects the plan-state of the plan `plan`.

Semantics specify, that once a plan reaches the state of *considered*, a case distinction is made, where the three possible cases are, that the filter condition of the plan is currently satisfied, it is not satisfied right now, but could be satisfied in the future and lastly, the filter condition is not satisfied and also can not be satisfied in the future. Dependent on this decision, control either advances to state *possible*, remains in state *considered* or aborts the execution by changing the state to *rejected*. This is followed by the program given in Figure 3(b). There the predicate *plan-not-filter* corresponds to the fact, that the filter condition is not satisfied and will stay so forever. The predicate *plan-filter* expresses the fact, that the filter condition is satisfied right now. In this paper we concentrate on showing how our synchronous parallel mechanism works and how the rely guarantee technique can be applied. Details of the Asbru verification in KIV in general are shown in [20].

3 Semantics of ITL+

In this section we give an informal overview over the temporal logic calculus we use. The calculus is integrated into KIV. The logical formalism is described in detail by Balser [15]. The temporal logic framework is a variant of ITL [21, 22] that is extended by explicitly including the behavior of the environment into each step. The basis for ITL are finite or infinite sequences π of valuations, which

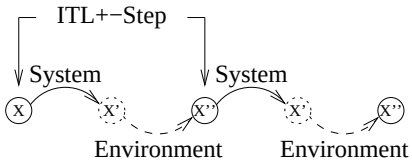


Figure 4: Interleaving of system and environment steps

are called *intervals*. Valuations in π are called *states*. Each state is described by a first-order predicate logic formula over dynamic variables V, which also can be *primed* V' or *double primed* V''. The relation between V and V' is called *system transition*, whereas the relation between V' and V'' *environment transition*. The value of V'' in a state must be equal to the value of V in the next successive state. Thereby system and environment transitions alternate (see Fig. 4). Constant variables are written in small letters. A small selection of temporal operators supported by KIV are:

$\Box\varphi$	φ holds *always* from now on in every state
φ **unless** ψ	either φ holds always from now on in every state or ψ holds in any state and φ holds in every state before
$X := t$	assignment
$\varphi_1; \varphi_2$	sequential composition
if ψ **then** φ_1 **else** φ_2	case distinction
while ψ **do** φ	loop

As shown, our ITL variant supports classic temporal logic operators as well as program operators. This allows us to mix programs with temporal logic formulas. Program operators describe only system steps that alternate with arbitrary environment steps.

A typical sequent in proofs about programs has the form $P, A, \Gamma \vdash \Delta$. P is the interleaved program that executes the system steps, A contains a temporal formula that describes behavior of the environment and Γ is a first order formula for the current variable assignment, while Δ contains the property which has to be shown. To verify that Δ holds in this sequent it must be shown that the current state Γ does not violate Δ and that the rest of the program run of P does not violate Δ either. To show the second part, symbolic execution is used.

For example, a sequent of the form mentioned above might look like this:

$$M := M + 1; P',\ \Box\ M' = M'', M = 2\ \vdash\ \Box\ M > 0$$

The program executed is $M := M + 1; P'$ (here, P' is a placeholder for the remaining SPL program) and the environment is assumed not to change M (formula $\Box\ M' = M''$). As the current state $M = 2$ does not violate $\Box\ M > 0$, a symbolic execution step is used to show that the rest of the program does not violate that formula too. The intuitive idea of a symbolic execution step is to

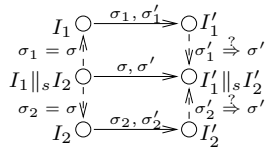


Figure 5: Schematics of parallel execution

execute the first program statement, i.e. applying the changes on the current state and to discard the first statement. So for the example above, a symbolic execution step would lead to the following formula:

$$P',\ \Box\ M' = M'', M = 3\ \vdash\ \Box\ M > 0$$

Of course, the environment assumption has to be considered too, but it simply leaves M unchanged in this example. To show that the sequent is valid, symbolic execution of the remaining program P' must be continued until P' terminates or induction can be applied. More complex formulas in the succedent might change during the step too (e.g. if the formula in the succedent is a program too, it has to be symbolically executed like the example program in the antecedent).

The basic idea to prove safety properties is to advance in the interval until a valuation that was considered earlier in the interval is reached. In this case a loop was executed. If we can prove that the property is true before and during the loop so it is invariant, then the proof can be finished with an inductive argument.

4 Synchronous Parallel Execution

The semantics of Asbru is defined as the synchronous parallel execution of all running plans. One task implementing Asbru in ITL+ was therefore defining a synchronous parallel execution mechanism to be embedded in ITL+.

The main issue for the definition of synchronous parallel execution is how to deal with multiple updates to the same variable. In Fig. 5 this has been visualized. Assuming the initial state is designated σ and the system description is $I_1 \| _s I_2$, it has to be defined what the resulting state of the execution of this system, σ', is. With the principle of recursion and the definition of the execution of sequential programs, the state changes caused by the individual I_1 and I_2 can be assumed to be given.

In the case where two processes assign values to different variables, synchronous execution simply executes both assignments. E.g. for the following program

$$M := 1; \alpha \| _s N := 2; \beta$$

the resulting state for the variables should be M' = 1 and N' = 2.

However, symbolic execution of a synchronous system gets more difficult when clashes occur, i.e. different processes write to the same variable within the same step. A mechanism has to be found how to solve these conflicts and how to describe the resulting state. To complicate matters, different behaviors might be relevant in different case studies. Certain case studies might require such parallel assignments to lead to indeterministic results. Other case studies might require the result to be indeterministically chosen from the two assignments made. And again, other case studies might require a completely different behavior.

To accommodate all these different requirements, it is impossible to specify the resolution of clashes on the level of the implementation of the synchronous parallel execution operator. Instead, the handling of clashes in KIV can be specified on case study level. For this, a predicate *sync* is used for every data type *type* in the implementation with the following syntax:

$sync : type \times type \times type \times type \mapsto bool$

This predicate is left unspecified at the implementation level but has to be specified at a case study level. This way, it is possible to formulate all of the above mentioned ways of dealing with concurrent assignments, even different ways for different data types in the same case study.

So e.g. the sequent

$$N := 1; \alpha \|_s N := 2; \beta,\ N = 0,\ \Box\ N'' = N',\ \Gamma\ \vdash\ \Delta$$

where both processes write on variable N leads to the following sequent after a symbolic execution step

$$\alpha \|_s \beta,\ sync(0, 1, 2, n),\ N = n, \Box\ N'' = N',\ \Gamma\ \vdash\ \Delta$$

The value of N after this clash depends entirely on the definition of the *sync*-predicate. The first parameter of the *sync*-predicate contains the value before the step, while the second and third parameter represent the values assigned by the left and right process. The forth parameter contains the value to N be assigned to the variable after execution.

4.1 Properties of the Synchronous Parallel Operator

Commutativity and associativity of the $\|_s$ operator depend on the definition of the *sync*-predicate. Two properties may be used to test, whether the *sync*-predicate is associative and commutative:

sync(a, b, c, d) ∧ *sync*(a, c, b, e) → d = e
sync(a, b, c, e) ∧ *sync*(a, e, d, f) ∧ *sync*(a, b, g, h) ∧ *sync*(a, c, d, g) → h = f

If the first property holds for *sync*, the $\|_s$ operator is commutative, while the second property must hold for associativity. All *sync*-predicates we use in this paper satisfy both properties, therefore we can use associativity and commutativity for the $\|_s$ operator.

Furthermore, the synchronous parallel operator we use is compositional, i.e. the following rule can be used

$$\frac{\vdash\ \varphi_1 \rightarrow \varphi_2 \qquad \varphi_2 \|_s \psi,\ \Gamma\ \vdash\ \Delta}{\varphi_1 \|_s \psi,\ \Gamma\ \vdash\ \Delta} \qquad \text{(comp)}$$

With this rule we are able to replace a component φ_1 with a more abstract component φ_2, if $\varphi_1 \rightarrow \varphi_2$ holds. A similar rule can be constructed for replacing the sub-formula ψ. We use this rule for abstraction of subcomponents as well as for applying the modularization theorem.

4.2 Dealing with Clashes in Asbru

The most important data structure in the Asbru case study is a dynamic function, mapping plan-names to plan-states. Every Asbru plan, which is active, updates this function every step with its current plan-state. These updates constitute concurrent write access to this dynamic function. As every place of the dynamic function is only updated once every step, the clashes can be resolved.

The dynamic function is defined as follows:

asbru-state : plan-name $\mapsto$ plan-state

For this data-structure, the syntax and semantics of the *sync*-predicate have to be defined. The syntax is given by the following definition:

sync : *asbru-state* $\times$ *asbru-state* $\times$ *asbru-state* $\times$ *asbru-state* $\mapsto$ *bool*

The semantics is defined by the following equation, assuming that as, as_0, as_1, as_2 are all of type *asbru-state*:

$$\begin{aligned} \mathit{sync}(\text{as}, \text{as}_1, \text{as}_2, \text{as}_0) \quad \leftrightarrow \quad \forall\, \text{a}.\ & \text{as[a]} = \text{as}_1\text{[a]} \wedge \text{as}_2\text{[a]} = \text{as}_0\text{[a]} \\ & \vee \text{as}_0\text{[a]} = \text{as}_1\text{[a]} \wedge \text{as}_2\text{[a]} = \text{as[a]} \\ & \vee \text{as}_1\text{[a]} = \text{as}_2\text{[a]} \wedge \text{as}_2\text{[a]} = \text{as}_0\text{[a]}; \end{aligned}$$

The first conjunct of the right hand side handles the case where first process did not change the value of as[a], therefore the value as_2[a], assigned by the second process, is used as the value of the resulting as_0[a]. The second conjunct is analogous, only with both processes swapped. The last conjunct handles the case if both processes assign the same value, which is taken as new value then. All other possibilities of assignment are considered invalid and lead to a clash.

5 Modularization Theorem

A popular technique for verification of parallel components is the rely-guarantee (R/G) paradigm.

To show that a system fulfills a global guarantee the parallel system is decomposed into its components. For each of these component a suitable local

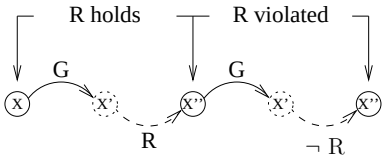


Figure 6: Example for Rely/Guarantees

guarantee and a rely must be specified. The component is obliged not to violate this guarantee as long as the environment, which is generally made of the other components, does not violate the rely. Finally, it must be shown that the global guarantee follows from the local guarantees and that no local guarantee violates the rely of any other component. Usually it is very difficult to come up with the exact proof obligations to show all these properties, therefore often a modularization theorem is used, that gives all proof obligations needed to establish the global guarantee and all local assumptions.

One of the difficulties of modularization is to avoid circularity. For this, Lamport [23] has proposed the $\xrightarrow{+}$ operator. The informal semantics of a formula $R \xrightarrow{+} G$ is, that a guarantee G has to hold one step longer then the rely R.

We specify assumptions and guarantees with predicates. These define a relation between primed and unprimed versions of the variables. This approach is similar to e.g. the formalization of rely/guarantees in [24]. This allows us to use the ITL **unless** operator as substitution for the $\xrightarrow{+}$ operator, i.e.

$$R(V', V'') \xrightarrow{+} G(V, V') \;:=\; G(V, V') \textbf{ unless } (G(V, V') \wedge \neg\, R(V', V''))$$

This formula states that G must hold in all steps before R is violated for the first time. The occurrence of G on the right hand of the **unless** operator is necessary, as system steps come before environment steps (see Fig. 5).

This operator allows us to construct the following modularization rule:

$$\frac{\begin{array}{ll} \mathit{sync}(n, n_1, n_2, n_3), G_1(n, n_1), G_2(n, n_2) \vdash G(n, n_3) & (1) \\ \mathit{sync}(n, n_1, n_2, n_3), R(n_3, n_4), G_1(n, n_1), G_2(n, n_2) & \\ \qquad \vdash R_1(n_1, n_4) \wedge R_2(n_2, n_4) & (2) \\ G_1(n, n_1) \vee G_2(n, n_1) \vdash G(n, n_1) & (3) \end{array}}{\begin{array}{c} R_1(N', N'') \xrightarrow{+} G_1(N, N') \;\; ||_S \;\; R_2(N', N'') \xrightarrow{+} G_2(N, N') \\ \vdash R(N', N'') \xrightarrow{+} G(N, N') \end{array}}$$

In the first premise, it is shown that both guarantees G_1 and G_2 preserve the overall guarantee G. The second premise shows, that R_1 and R_2 are preserved by both guarantees. The last premise is necessary for the case when one process terminates. Here it is shown that both single premises preserve G.

This theorem was proven in KIV by symbolic executing of the conclusion, while the three premises were assumed as lemmas. After the first step, three cases

occur. In the first two cases, either the left or the right component has terminated. Both cases could be resolved by using the third premise and induction after an additional step. In the third case, both components are still running. Premise (1) and (2) of the theorem can be used to show that G, R_1 and R_2 still hold and the branch can be close by induction.

6 Application

In the Asbru case study we are using the techniques described above for the verification of medical guidelines. The breast cancer case study consists of almost 50 plans, which is a size, which cannot be handled with simple symbolic execution alone. The solution is to use rely guarantee properties for some of the plans to contain the indeterminism of these plans. With the composition theorem it is then possible to combine these abstractions. In this section we will illustrate our approach on an example from the breast cancer guideline.

For this we are looking at a medically relevant property, designated as property 20. This property states, that a radiotherapy to the chest wall is unwanted in breast cancer patients which had their axilla nodes removed surgically. To formalize this property, it has to be linked to the breast-cancer case-study first. Examination of the case-study reveals that both the radiotherapy to the chest wall and the removal of the axilla nodes can be located in a subsection of the case-study. This subsection has already been presented in Figure 1 and explained in Section 2. The Asbru-plan designated `cand` in the figure corresponds to a removal of the axilla nodes. The second interesting plan is designated `praoa` in the figure. This plan describes the administration of radiotherapy to the axilla.

The medical property can be formalized, such that plan `praoa` must not be started once the plan `cand` has been started. Next, it is necessary to find abstractions for all plans in that hierarchy, such that the overall properties can be verified using these abstractions. In this paper we will present ideas for all the sub-properties required for the verification, however, we will only describe the most important sub-property and its verification obligations in detail.

The top-level plan of this subsection of the guideline is specified such that only one of its sub-plans is activated at a time. The plan to be activated is chosen in an arbitrary order. Therefore, it is impossible, that plan `cand` and plan `toan` are activated at the same time. With plan `toan` being the super-plan of plan `praoa` it follows, that plan `cand` and plan `praoa` cannot be activated at the same time. This result is independent of all the other plans in the hierarchy.

In a next step, it has to be evaluated if both plans can be activated one after the other. This is also impossible. Plan `dwa` activates its sub-plans one after another until one of the sub-plans reaches the completed state. With one completed sub-plan, plan `dwa` terminates itself. The structure of the plan `cand` implies, that this plan will eventually complete, once it has been activated. Similarly

the `toan` plan can only complete once it is activated. Therefore if one of the relevant plans is activated, it follows, that the other plan will not be activated subsequently.

This implies that the behavior of all the other plans of this hierarchy is completely unimportant with respect to the correctness of the property 20. However this is only partially true. Although the sub-plans of the top-level plan `dwa` are activated sequentially, the representing plan procedures are still all executed in parallel. If their behavior is abstracted to be completely arbitrary it cannot be ruled out that their updates to the global state results in clashes, as described in Section 4. For all these plans it has to be verified that their updates to the global state are non-conflicting with the updates of other plans.

Next is the formulation of the properties for the plans `cand` and `toan` (as the super-plan of `praoa`). The relevant behavior of the `cand` plan is that it gets activated only when the super-plan `dwa` sends the activation signal. Apart from that the plan will always complete and may not be aborted or rejected. An exception here is the case, when the super-plan `dwa` itself terminates, in which case `cand` might also get rejected or aborted. Also it is necessary to formalize that the plan will not leave the completed state once it completes.

$$
\begin{aligned}
&\mathrm{G}_{cand} \equiv \\
&\quad (\quad \mathrm{AS}[\text{`cand'}] \neq \text{aborted} \wedge \mathrm{AS}[\text{`cand'}] \neq \text{rejected} \wedge \neg\ \text{terminatedP}(\mathrm{AS}[\text{dwa}]) \\
&\quad\ \rightarrow\ \mathrm{AS'}[\text{`cand'}] \neq \text{aborted} \wedge \mathrm{AS'}[\text{`cand'}] \neq \text{rejected}) \\
&\wedge\ (\mathrm{AS}[\text{`cand'}] = \text{completed} \rightarrow \mathrm{AS'}[\text{`cand'}] = \text{completed}) \\
&\wedge\ (\text{evaluatedP}(\mathrm{AS}[\text{`cand'}]) \wedge \neg\ \mathrm{PC}[\text{`cand'}] \rightarrow \text{evaluatedP}(\mathrm{AS'}[\text{`cand'}]))
\end{aligned}
$$

This has been formalized as in the formula above. This formula describes the guarantee of the `cand` plan. The dynamic function storing the plan-states of all asbru-plans is designated as AS. The selector AS[‘cand’] selects the plan-status of the `cand` plan.

First part of this formula states that `cand` will not change its state to aborted or rejected as long as plan-state of the super-plan `dwa` is not terminated. Second part of the conjunction of this property requires the plan not to change its plan-state once the completed state has been reached.

The activation signals of the asbru-plans, which are designated as E in Figure 2, is stored in a dynamic function PC mapping plan-names to boolean variables. The third and final part of the conjunction of the guarantee requires the plan to stay in the evaluation phase until the activation signal is received.

For this guarantee to hold, the plan relies on certain behavior of the environment. For one, the environment is not allowed to change the status of the plan. As every plan updates its own state this should always be true. The second part of the rely is, that the super-plan never enters the suspended state. Therefore the rely is formulated as follows:

$$\mathrm{R}_{cand} \equiv \mathrm{AS'}[\text{`cand'}] = \mathrm{AS''}[\text{`cand'}] \wedge \mathrm{AS''}[\text{dwa}] \neq \text{suspended}$$

With similar reasoning, a property for the `toan` plan can be found. This is necessary as the `dwa` plan has no possibility of directly interacting with the `praoa` plan. Instead, `dwa` influences the execution of the `toan` plan, which in turn controls the execution of `praoa`. Therefore, the guarantee for the `toan` plan is formulated as follows:

$G_{toan} \equiv$
(AS[‘toan’] ≠rejected ∧ ¬ terminatedP(AS[‘dwa’]) → AS’[‘toan’] ≠rejected)
∧ (AS[‘toan’] ≠aborted ∧ ¬ terminatedP(AS[‘dwa’]) → AS’[‘toan’] ≠aborted)
∧ (AS[‘toan’] = completed → AS’[‘toan’] = completed)
∧ (evaluatedP(AS[‘toan’]) ∧ ¬ PC[‘toan’] → evaluatedP(AS’[‘toan’]))
∧ (evaluatedP(AS[‘toan’]) ∧ AS[‘praoa’] ≠activated → AS’[‘praoa’] ≠activated)

This property mainly consists of the same components as the guarantee of the `cand` plan, with the only real difference, that the control of the activation of `praoa` is more complex. To cope with this it is simply stated that the plan `praoa` may not be activated as long as plan `toan` is evaluated. The rely of toan, R_{toan} is defined analogous to R_{cand}

To lift the guarantees from `cand` and `toan` the guarantees of both components have to be strengthened first

$$G'_{cand} := G_{cand} \wedge \text{frame}(AS, AS', \text{subplans}(cand))$$
$$G'_{toan} := G_{toan} \wedge \text{frame}(AS, AS', \text{subplans}(toan))$$

This strengthening requires `cand` and `toan` to restrict their behavior, such that the plans (and their respective sub-plans) do not change plan-status entries in the AS variable other than their own, which is formalized with the *frame* predicate.

As first step to lift the R/G properties from `cand` and `toan` components the following two properties are shown for these plans

$$\text{inactive\#(‘cand’)} \vdash R_{cand} \xrightarrow{+} G_{cand} \quad (1)$$
$$\text{inactive\#(‘toan’)} \vdash R_{toan} \xrightarrow{+} G_{toan} \quad (2)$$

For both proofs only a small set of plans have to be considered, which makes it feasible to do them by symbolic execution. In the next step the modularization theorem is used to show that the R/G properties of both components can be combined, i.e.

$$R_{cand} \xrightarrow{+} G_{cand} \parallel_s R_{toan} \xrightarrow{+} G_{toan} \vdash R_{cand} \wedge R_{toan} \xrightarrow{+} G'_{cand+toan} \quad (3)$$

where

$$G'_{cand+toan} := G_{cand} \wedge G_{toan} \wedge \text{frame}(AS, AS', \text{subplans}(cand)+\text{subplans}(cand))$$

The verification of the three predicate proof obligations given by the theorem is straight forward. As final step we show the overall R/G property for both plans

$$\text{inactive\#(‘cand’)} \parallel_s \text{inactive\#(‘toan’)} \vdash R_{cand} \wedge R_{toan} \xrightarrow{+} G'_{cand+toan}$$

To show this property, compositionality is applied with formulas (1) and (2). This leads to the same sequent as (3) which was already shown.

This strategy can be applied for all other components up to the top level plan. By using this strategy the complexity and indeterminism of all sub-plans of `toan` and `cand` plans can be contained in a separate proof. The remaining sub-plans of the `dwa` plan can be dealt with using standard abstractions, which are already predefined. The verification of twelve plans can therefore be reduced to the verification of one plan and one tl-formula in parallel plus a number of smaller, much more manageable separate proofs. As the complexity of these proofs is exponentially in the number of parallel plans, the effort spent verifying all auxiliary proofs does not add up to the complexity of direct symbolic execution.

7 Conclusion and Related Work

In this paper we suggest a generic solution how to deal with synchronous parallel write access to variables which enables the use of dynamic functions as a common store for several process for data exchange. This is implemented in the fully compositional ITL+ temporal logic. Compositionality of the synchronous parallel operator together with a modularization theorem allows us to use of the well known rely guarantee approach to abstract complex procedures to relatively simple temporal logic formulas. This enables us to deal with hierarchies up to 50 processes. To show that our approach allows to work with real-world medical properties and guidelines. we applied this methodology to verify the correctness of medical guidelines relative to medical properties.

The rely guarantee technique was first introduced in [13] and [14]. An important work about compositionality is Abadi and Lamport [23]. There, assumptions and guarantees can be specified as temporal logic formulas. For composition of components they use the conjunction operator, which allows synchronous as well as asynchronous execution. One major difference in their work is, that all formulas have to be specified in a special normal form, while our formalism allows the direct inclusion of various specification languages, which we use e.g. to formalize the asbru case-study. Another work about compositionality is by Cau et al [24]. They use a generic modularization theorem, that can be instantiated for arbitrary parallel operators fulfilling certain conditions. We believe that defining a synchronous parallel operator similar to ours in their framework might lead to a similar theorem. A work that applies compositionality and ITL on real world applications is Solanki et al [25]. However, their application (semantic web service descriptions) is quite different to ours, so they are difficult to compare. There are many other approaches that use the rely guarantee technique. An overview can be found in e.g. in [26]. To our knowledge this is the first approach that combines the rely guarantee paradigm with symbolic execution, which is considered as an intuitive proof method for interactive verification.

There are different approaches which try to apply formal methods to medical

guidelines. Scope of the group of John Fox is a simpler, more intuitive but also less expressive [27] language for guideline formalization called PRO*forma* [28]. The group around Paolo Terenziani follows the same idea as the protocure projects, namely a stepwise formalization of medical guidelines and finally a formal verification of the derived model. Currently this group [4] concentrates on the use of automatic model checking done with the SPIN[1] modelchecker. The work presented here is the first, that uses rely guarantee reasoning on medical guidelines.

References

[1] : Clinical Practice Guidelines: Directions for a New Program. National Academy Press (1992)

[2] : Implementing clinical practise guidelines. Effective Health Care Bulletin (1994) York: University of York.

[3] Clayton, P., Hripcsak, G.: Decision support in healthcare. International Journal of Bio-Medical Computing **39** (4 1995) 59–66

[4] Giordano, L., Terenziani, P., Bottrighi, A., Montani, S., Donzella, L.: Model Checking for Clinical Guidelines: an Agent-based Approach. In: AMIA Annu Symp Proc. (2006) 289–293

[5] Lucas, P.: Quality Checking of Medical Guidelines through Logical Abduction. In: Proc. of AI-2003, volume XX, Springer (2003) 309–321

[6] ten Teije, A., Marcos, M., Balser, M., van Croonenborg, J., Duelli, C., van Harmelen, F., Lucas, P., Miksch, S., Reif, W., Rosenbrand, K., Seyfang, A.: Improving medical protocols by formal methods. Artificial Intelligence in Medicine **36(3)** (3 2006) 193–209

[7] Seyfang, A., Kosara, R., Miksch, S.: Asbru 7.3 reference manual. Technical report, Vienna University of Technology (2002)

[8] Schmitt, J., Hoffmann, A., Balser, M., Reif, W., Marcos, M.: Interactive verification of medical guidelines. In Misra, J., Nipkow, T., Sekerinski, E., eds.: Formal Methods 2006, Proceedings. Volume 4085 of LNCS., Springer (2006) 32–47

[9] Bäumler, S., Balser, M., Dunets, A., Reif, W., Schmitt, J.: Verification of medical guidelines by model checking: a case study. In: Proc. of the 13th International SPIN Workshop on Model Checking of Software. Number 3925 in LNCS, Springer-Verlag (2006) 219–233

[10] Marcos, M., Roomans, H., ten Teije, A., van Harmelen, F.: Improving medical protocols through formalisation: a case study (2002)

[1] www.spinroot.com

[11] Marcos, M., Balser, M., ten Teije, A., Harmelen, F.V.: From informal knowledge to formal logic: a realistic case study in medical protocols

[12] Polo-Conde, C., Marcos, M., Seyfang, A., Wittenberg, J., Miksch, S., Rosenbrand, K.: Assessment of mhb: an intermediate language for the representation of medical guidelines. In: Proc. of the 10th Conference of the Spanish Association for Artificial Intelligence (CAEPIA-05). (2005) I–19 – I–28

[13] Misra, J., Chandi, K.: Proofs of networks of processes. IEEE Transactions of Software Engineering (1981)

[14] Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4) (1983) 596–619

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

[16] Balser, M., Reif, W., Schellhorn, G., Stenzel, K.: KIV 3.0 for Provably Correct Systems. In Hutter, D., Stephan, W., Traverso, P., Ullmann, M., eds.: Proc. Int. Wsh. Applied Formal Methods. Volume 1641 of LNCS., Springer (1999) 330–337

[17] Miksch, S., Shahar, Y., Johnson, P.: Medizinische Leitlinien und Protokolle: das Asgaard/Asbru Projekt. KI-Journal (1997) 34–37 Themenheft Medizin.

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

[19] Pnueli, A., Shalev, M.: What is in a step: On the semantics of statecharts. In: Symposium on Theoretical Aspects of Computer Software. Volume 526 of LNCS., Springer (1991) 244–264

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

[21] Moszkowski, B.: Executing Temporal Logic Programs. Cambridge University Press, Cambridge (1986)

[22] Cau, A., Moszkowski, B., Zedan, H.: ITL – Interval Temporal Logic. Software Technology Research Laboratory, SERCentre, De Montfort University, The Gateway, Leicester LE1 9BH, UK. (2002) http://www.cse.dmu.ac.uk/STRL/ITL/.

[23] Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems (1995)

[24] Cau, A., Collette, P.: Parallel composition of assumption-commitment specifications: A unifying approach for shared variable and distributed message passing concurrency. Acta Inf. **33**(2) (1996) 153–176

[25] Solanki, M., Cau, A., Zedan, H.: Augmenting semantic web service descriptions with compositional specification. In Feldman, S.I., Uretsky, M., Najork, M., Wills, C.E., eds.: Proc. of 13th int. conference on World Wide Web, ACM (2004) 544–552

[26] de Roever, W.P., et al.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge University Press (2001)

[27] Peleg, M., Tu, S., Bury, J., Ciccarese, P., Fox, J., Greenes, R., Hall, R., Johnson, P., Jones, N., Kumar, A., Miksch, S., Quaglini, S., Seyfang, A., Shortliffe, E., Stefanelli, M.: Comparing computer-interpretable guideline models: A case-study approach. JAMIA **10**(1) (2003)

[28] Fox, J., Patkar, V., Thomson, R.: Decision support for health care: the proforma evidence base. Inform Prim Care **14**(1) (2006) 49–54