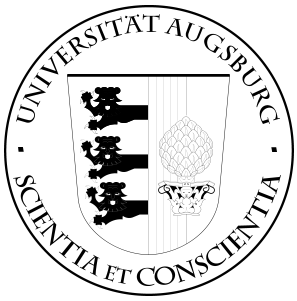
Universität Augsburg



Interactive Verification of Concurrent Systems using Symbolic Execution

M. Balser, W. Reif

Report 2008-12 July 2008

Institut für Informatik
D-86135 Augsburg

Abstract

This technical report presents an interactive proof method for the verification of temporal properties of concurrent systems based on symbolic execution. Symbolic execution is a well known and very intuitive strategy for the verification of sequential programs. We have carried over this approach to the interactive verification of arbitrary linear temporal logic properties of (infinite state) parallel programs. The resulting proof method is very intuitive to apply and can be automated to a large extent. It smoothly combines first order reasoning with reasoning in temporal logic. The proof method has been implemented in the interactive verification environment KIV and has been used in several case studies.

Contents

1 Introduction

Compared to sequential programs, the design and not only the verification of concurrent systems is more difficult, mainly because the control flow is more complex. Particularly, for reactive systems, not only the final result of execution but the sequence of output over time is relevant to system behavior. Finding errors by means of testing strategies is limited, because, especially for interleaved systems, an exponential amount of possible executions must be considered. The execution is nondeterministic, making it difficult to reproduce errors. An alternative to testing is the use of formal methods to specify and verify concurrent systems with mathematical rigor. Automatic methods – especially model checking – have been applied successfully to discover flaws in the design and implementation of systems. Starting from systems with finite state spaces of manageable size, research in model checking aims at mastering ever more complex state spaces and to reduce infinite state systems by abstraction. In general, systems must be manually abstracted to ensure that formal analysis terminates. An alternative approach to large or infinite state spaces are interactive proof calculi. They directly address the problem of infinite state spaces. Here, the challenge is rather to keep track of the proofs and to achieve a high degree of automation. Existing interactive calculi to reason in temporal logic about concurrent systems are generally difficult to apply. The strategy of symbolic execution, on the other hand, has been successfully applied to the interactive verification of sequential programs (e.g. Dynamic Logic [7, 9]). Symbolic execution gives intuitive proofs with a high degree of automation.

Combining Dynamic Logic and temporal logic has already been investigated, e.g. Process Logic [15] and Concurrent Dynamic Logic [14] and more recently [6, 8]. These works focus on combinations of logic, while we are interested in an interactive proof method based on symbolic execution. Symbolic execution of parallel programs has been investigated in [1], however, this approach has been restricted to the verification of pre/post conditions. Other approaches are often restricted to the verification of certain types of temporal properties. Our approach presents an interactive proof method to verify arbitrary temporal properties for parallel programs with the strategy of symbolic execution and thus promises to be intuitive and highly automatic.

Our logic is based on Interval Temporal Logic (ITL) [13]. The chop operator $\varphi; \psi$ of ITL corresponds to sequential composition and programs are just a special case of temporal formulas. In addition, we have defined an interleaving operator $\varphi \parallel \psi$ to interleave arbitrary temporal formulas. Furthermore, our logic explicitly considers arbitrary environment steps after each system transition, which is similar to Temporal Logic of Actions (TLA) [11]. This ensures that systems are compositional and proofs can be decomposed. In total, we have defined a compositional logic which includes a rich programming language with interleaved parallel processes similar to the one of STeP [5]. Important for interactive proofs, system descriptions need not be translated to flat transition systems, the logic rather offers advanced programming constructs and the calculus directly reasons about programs.

In this paper, we focus on the strategy of symbolic execution of concurrent systems and show how to verify safety and liveness properties with the same proof

strategy. Our claim is that with symbolic execution, interactive verification in temporal logic becomes very intuitive in practice.

A short overview of our logic is given in Section 2. The calculus for symbolic execution is described in Section 4. The strategy has been implemented in KIV (see Section 5). As an example we consider an algorithm to calculate the Binomial coefficient [12] (see Sections 3 and 6). Section 7 concludes. For more details on the logic and calculus, especially on induction and double primed variables to decompose proofs, we refer to [2].

2 Logic

Similar to [13], we have defined a first order interval temporal logic with static variables a, dynamic variables A, functions f, and predicates p. Let v be a static or dynamic variable. Then, the syntax of (a subset of) our logic is defined

$$\begin{array}{rcl} e & ::= & a \mid A \mid A' \mid A'' \mid f(e_1, \ldots, e_n) \\ \varphi & ::= & p(e_1, \ldots, e_n) \mid \neg\, \varphi \mid \varphi_1 \wedge \varphi_2 \mid \exists\, v.\, \varphi \\ & & \mid \textbf{step} \mid \varphi_1; \varphi_2 \mid \varphi^* \\ & & \mid \lceil A_1, \ldots, A_n \rceil \mid \varphi_1 \mid\mid^{<} \varphi_2 \end{array}$$

Dynamic variables can be primed and double primed. It is possible to quantify both static and dynamic variables. The chop operator $\varphi_1; \varphi_2$ directly corresponds to the sequential composition of programs. The star operator φ^* is similar to a loop. We have added an operator $\lceil A_1, \ldots, A_n \rceil$ to define an explicit frame assumption. Furthermore, operator $\varphi_1 \mid\mid^{<} \varphi_2$ can be used to interleave two "processes". The basic operator $\mid\mid^{<}$ gives precedence to the left process, i.e., a transition of φ_1 is executed first.

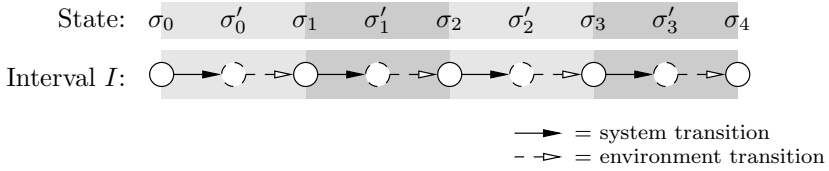


Figure 1: An interval as sequence of states

In ITL, an interval is considered to be a finite or infinite sequence of states, where a state is a mapping from variables to their values. In our settings, we introduce additional intermediate states σ'_i to distinguish between system and environment transitions. For intuition, compare the following to Figure 1. Let $\overline{n} \in \mathbb{N}^\infty$. An *interval*

$$I = (\sigma_0, \sigma'_0, \sigma_1, \ldots, \sigma'_{\overline{n}-1}, \sigma_{\overline{n}})$$

consists of an initial state σ_0, and a finite or infinite and possibly empty sequence of transitions $(\sigma'_i, \sigma_{i+1})_{i=0}^{\overline{n}-1}$. In the intermediate states σ'_i the values of the variables after a system transition are stored. The following states σ_{i+1} reflect the states after an environment transition. In this manner, system and environment transitions alternate. An empty interval consists only of an initial state σ_0.

Given an interval I, variables are evaluated as follows:

$$\begin{array}{rcl} [\![v]\!]_I & := & \sigma_0(v) \\ [\![A']\!]_I & := & \begin{cases} \sigma'_0(A) & \text{if } |I| > 0 \\ \sigma_0(A) & \text{otherwise} \end{cases} \\ [\![A'']\!]_I & := & \begin{cases} \sigma_1(A) & \text{if } |I| > 0 \\ \sigma_0(A) & \text{otherwise} \end{cases} \end{array}$$

Static and unprimed dynamic variables are evaluated in the initial state. Primed variables are evaluated in σ'_0 and double primed variables in σ_1. In the last state,

i.e. if I is empty, the value of a primed or double primed variable is equal to the unprimed variable. It is assumed that after a system has terminated, the variables do not change.

The semantics of the standard ITL operators can be carried over one-to-one to our notion of an interval, and is therefore omitted here. In [13], however, assignments only restrict the assigned variables, whereas program assignments leave all other dynamic variables unchanged. This is known as a frame assumption. In our logic, we have defined an explicit frame assumption $\lceil A_1, \ldots, A_n \rceil$ which states that a system transition leaves all but a selection of dynamic variables unchanged.

$$I \models \lceil A_1, \ldots, A_n \rceil \quad \text{iff} \quad \sigma_0'(A) = \sigma_0(A) \text{ for all } A \notin \{A_1, \ldots, A_n\}$$

Details on frame assumptions can be found in [2].

The interleaving operator $\varphi \mid\mid^< \psi$ interleaves arbitrary formulas φ and ψ. The two formulas represent sets of intervals. We have therefore defined the semantics of $\varphi \mid\mid^< \psi$ relative to the interleaving $[\![I_1 \mid\mid^< I_2]\!]$ of two concrete intervals I_1 and I_2.

$$I \models \varphi \mid\mid^< \psi \quad \text{iff} \quad \begin{array}{l}\text{there exist } I_1, I_2 \\ \text{with } I \in [\![I_1 \mid\mid^< I_2]\!] \text{ and } I_1 \models \varphi \text{ and } I_2 \models \psi\end{array}$$

For nonempty intervals $I_1 = (\sigma_0, \sigma_0', \sigma_1, \ldots)$, and $I_2 = (\tau_0, \tau_0', \tau_1, \ldots)$, interleaving of intervals adheres to the following recursive equations.

$$[\![I_1 \mid\mid^< I_2]\!] = \begin{cases} (\sigma_0, \sigma_0') \oplus [\![(\sigma_1, \ldots) \parallel I_2]\!], & \text{if } I_1 \text{ is not blocked} \\ (\tau_0, \tau_0') \oplus [\![(\sigma_1, \ldots) \parallel (\tau_1, \ldots)]\!], & \text{if } I_1 \text{ is blocked, } \sigma_0 = \tau_0 \\ \emptyset, & \text{otherwise} \end{cases}$$

$$[\![I_1 \,\big|\big|\, I_2]\!] = [\![I_1 \mid\mid^< I_2]\!] \cup [\![I_2 \mid\mid^< I_1]\!]$$

Interval I_1 is blocked, if $\sigma_0'(blk) \neq \sigma_0(blk)$ for a special dynamic variable blk. The system transition toggles the variable to signal that the process is currently blocked (see **await** statement below). If I_1 is not blocked, then the first transition of I_1 is executed and the system continues with interleaving the remaining interval with I_2. (Function $\oplus$ prefixes all of the intervals of a given set with the two additional states.) If I_1 is blocked, then a transition of I_2 is executed instead. However, the blocked transition of I_1 is also consumed. A detailed definition of the semantics can be found in [2].

Additional common logical operators can be defined as abbreviations. A list of frequently used abbreviations is contained in Table 1, where most of the abbreviations are common in ITL. The next operator comes in two flavors. The strong next $\circ \varphi$ requires that there is a next step satisfying φ, the weak next $\bullet \varphi$ only states that if there is a next step, it must satisfy φ.

As can be seen in Table 1, the standard constructs for sequential programs can be derived. Executing an assignment requires exactly one step. The value of e is "assigned" to the primed value of A. Other variables are unchanged ($\lceil A \rceil$). Note that for conditionals and loops, the condition ψ evaluates in a single step. For local variable definitions, the local variable is quantified ($\exists\, A$), in addition, the environment cannot access the local variable ($\Box\, A'' = A'$). The global value of the variable is unchanged ($\Box\, A' = A$).

$$
\begin{array}{rcl@{\qquad}rcl}
\textbf{more} & :\equiv & \textbf{step}; \text{ true} & \Diamond\, \varphi & :\equiv & \textbf{finite}; \varphi \\
\textbf{last} & :\equiv & \neg\, \textbf{more} & \Box\, \varphi & :\equiv & \neg \Diamond \neg\, \varphi \\
\textbf{inf} & :\equiv & \text{true}; \text{ false} & \circ\, \varphi & :\equiv & \textbf{step}; \varphi \\
\textbf{finite} & :\equiv & \neg\, \textbf{inf} & \bullet\, \varphi & :\equiv & \neg \circ \neg\, \varphi
\end{array}
$$

$$
\begin{array}{rcl}
A := e & :\equiv & A' = e \wedge \lceil A \rceil \wedge \textbf{step} \\
\textbf{skip} & :\equiv & \lceil\, \rceil \wedge \textbf{step} \\
\textbf{if } \psi \textbf{ then } \varphi_1 \textbf{ else } \varphi_2 & \equiv & \psi \wedge (\textbf{skip}; \varphi_1) \vee \neg\, \psi \wedge (\textbf{skip}; \varphi_2) \\
\textbf{while } \psi \textbf{ do } \varphi & :\equiv & ((\psi \wedge (\textbf{skip}; \varphi))^* \wedge \Box\, (\textbf{last} \rightarrow \neg\, \psi)); \textbf{skip} \\
\textbf{var } A \textbf{ in } \varphi & :\equiv & \Box\, A' = A \wedge \exists\, A.\, \varphi \wedge \Box\, A'' = A' \\
\textbf{bskip} & :\equiv & \mathit{blk}' \neq \mathit{blk} \wedge \lceil \mathit{blk} \rceil \wedge \textbf{step} \\
\textbf{await } \varphi \textbf{ do } \psi & :\equiv & ((\neg\, \varphi \wedge \textbf{bskip})^* \wedge \Box\, (\textbf{last} \rightarrow \varphi)); \psi \\
\textbf{await } \varphi & :\equiv & \textbf{await } \varphi \textbf{ do skip} \\
\varphi \parallel \psi & :\equiv & \varphi \parallel^{<} \psi \vee \psi \parallel^{<} \varphi
\end{array}
$$

Table 1: Frequently used temporal abbreviations

Parallel programs communicate using shared variables. In order to synchronize execution, the operator **await** ψ **do** φ can be used. A special dynamic variable *blk* is used to mark whether a parallel program is blocked. The **await** operator behaves like a loop waiting for the condition to be satisfied. While the operator waits, no variable is changed except for variable *blk* which is toggled. In other words, a process guarded with an **await** operator actively waits for the environment to satisfy its condition. Immediately after condition ψ is satisfied, construct φ is executed.

3 Example

The algorithm of Figure 2 is similar to the algorithm in [12] to calculate the binomial coefficient $B = \binom{n}{k}$. It illustrates that our programming language closely resembles the Simple Programming Language (SPL) of STeP. The procedure Binom$(n, k;\ \mathrm{b})$ with value parameters n, k calculates the binomial coefficient

$$B = \binom{n}{k} = \frac{n!}{k! * (n-k)!} = \frac{n * (n-1) * \ldots * (n-k+1)}{1 * 2 * \ldots * k}$$

using two parallel processes. After the variables have been initialised, the first process takes care of the numerator and the second process of the denominator. Separate assignments are used for read and write access to variable B, the intermediate result being stored in local variables T. Access to the shared variable B is synchronised with a semaphore S. The semaphore is aquired in sub procedure P$(; S)$ and released in V$(; S)$. An additional guard **await** $Y_1 + Y_2 \leq n$ ensures that the remainder of the divisions B/Y_2 in the second process is always 0. In Fig. 2, numbers have been added to refer to program positions in Section 6. The numbers are only used for illustration purposes.

We have verified two temporal properties

$$k \leq n \wedge \text{Binom}(n, k; B) \rightarrow \Diamond\, \textbf{last} \tag{1}$$

$\mathrm{Binom}(n, k; B)$
$\leftrightarrow$ **var** S, Y_1, Y_2 **in begin**
1: $S := 1$; 2: $Y_1 := n$; 3: $Y_2 := 1$; 4: $B := 1$;

5: **while** $n - k < Y_1$ **do begin**	5: **while** $Y_2 \leq k$ **do begin**
6: $\mathrm{P}(; S)$;	6: **await** $Y_1 + Y_2 \leq n$;
var T **in begin**	7: $\mathrm{P}(; S)$;
7: $T := B * Y_1$;	**var** T **in begin**
8: $B := T$	8: $T := B/Y_2$;
end	9: $B := T$
9: $\mathrm{V}(; S)$;	**end**
10: $Y_1 := Y_1 - 1$	10: $\mathrm{V}(; S)$;
end	11: $Y_2 := Y_2 + 1$
	end

where

$$\begin{aligned} \mathrm{P}(; S) &\leftrightarrow 1: \textbf{await } S > 0 \textbf{ do } S := S - 1 \\ \mathrm{V}(; S) &\leftrightarrow 1: S := S + 1 \end{aligned}$$

Figure 2: Algorithm to calculate the binomial coefficient

and

$$k \leq n \wedge \mathrm{Binom}(n, k; B) \wedge \Box\ B'' = B' \rightarrow \Box \left(\textbf{last} \rightarrow B = \binom{n}{k} \right) . \qquad (2)$$

The first property states that the algorithm eventually terminates and the second property ensures that the final result is correct. For both properties, we have to assume that $k \leq n$. The second property requires an additional environment assumption $\Box\ B'' = B'$ to ensure that the environment transition – as the relation between primed and double primed variables – does not modify variable B.

The example illustrates that program constructs are first order citizens in our logic. Parallel programs are just a special case of temporal formulas. Both safety and liveness properties can be expressed. In the example, interaction with the environment is trivial: we assume that the environment does not interfere. More complex environment interaction can be expressed.

4 Calculus

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

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

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

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

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

4.1 Normal form

Our proof strategy is to symbolically execute temporal formulas, parallel programs being just a special case thereof. In Dynamic Logic, the leading assignment of a DL formula $\langle v := e; \alpha \rangle\ \varphi$ is executed as follows:

$$\frac{\dfrac{\Gamma_v^{v_0}, v = e_v^{v_0} \vdash \langle \alpha \rangle\ \varphi}{\Gamma \vdash \langle v := e \rangle\ \langle \alpha \rangle\ \varphi}\ \mathit{asg\ r}}{\Gamma \vdash \langle v := e; \alpha \rangle\ \varphi}\ \mathit{normalize}$$

The sequential composition is normalized and is replaced with a succession of diamond operators. Afterwards, the assignment is "executed" to receive $\Gamma_v^{v_0}, v = e_v^{v_0}$ as weakest precondition. In our settings, we normalize 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, a program (or temporal formula) is rewritten to a formula of the following type

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

with τ being a formula in predicate logic describing a transition as a relation between unprimed, primed and double primed variables. In general, a program 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 a_i which occur in both τ_i and φ_i. The general pattern to separate the first transitions of a given temporal formula is

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

We will refer to this general pattern as normal form.

$$\frac{\varphi, \Gamma \vdash \Delta \quad \psi, \Gamma \vdash \Delta}{\varphi \vee \psi, \Gamma \vdash \Delta}\ \mathit{dis\ l} \qquad \frac{\varphi_a^{a_0}, \Gamma \vdash \Delta}{\exists\ a.\ \varphi, \Gamma \vdash \Delta}\ \mathit{ex\ l}$$

where a_0 fresh with respect to $\mathrm{free}(\varphi) \setminus \{a\} \cup \mathrm{free}(\Gamma, \Delta)$

$$\frac{\tau_{A,A',A''}^{a,a,a} \vdash}{\tau, \mathbf{last} \vdash}\ \mathit{lst} \qquad \frac{\tau_{A,A',A''}^{a_1,a_2,A}, \varphi}{\tau, \circ\ \varphi \vdash}\ \mathit{stp}$$

where a, a_1, a_2 fresh with respect to $\mathrm{free}(\tau, \varphi)$

Table 3: Rules for executing an overall step

4.2 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.)

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

With the two rules *dis l* and *ex l* of Table 3, 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 a. The result is a formula in pure predicate logic with static variables only, which can be proven with standard first order reasoning. Rule *stp* advances a step in the trace. The values of the dynamic variables A and A' in the old state are stored in fresh static variables a_1 and a_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.

alw:	$\Box\,\varphi$	$\leftrightarrow$	$\varphi \wedge \bullet\,\Box\,\varphi$
ev:	$\Diamond\,\varphi$	$\leftrightarrow$	$\varphi \vee \circ\,\Diamond\,\varphi$

Table 5: Rules for executing temporal logic operators $\Box$ and $\Diamond$

4.3 Executing temporal logic

The idea of symbolic execution can be applied to formulas in temporal logic. For example, operator $\Box\,\varphi$ is similar to a loop in a programming language: formula φ is executed in every step. An appropriate rewrite rule is *alw* of Table 5. Formula φ must hold now, and in the next step $\Box\,\varphi$ again holds. To arrive with a formula in normal form, the first conjunct of the resulting formula $\varphi \wedge \bullet\,\Box\,\varphi$ must be further rewritten. The rewrite rule above corresponds to the recursive definition of $\Box\,\varphi$. Other temporal operators can be executed similarly.

$$\frac{\vdash \varphi_1 \rightarrow \varphi_2}{\vdash \varphi_1; \psi \rightarrow \varphi_2; \psi} \; \textit{chp lem}$$

chp dis:	$(\varphi_1 \vee \varphi_2); \psi$	$\leftrightarrow$	$\varphi_1; \psi \vee \varphi_2; \psi$
chp ex:	$(\exists\, a.\ \varphi); \psi$	$\leftrightarrow$	$\exists\, a_0.\ \varphi_a^{a_0}; \psi$
chp lst:	$(\tau \wedge \mathbf{last}); \psi$	$\leftrightarrow$	$\tau^{A,A}_{A',A''} \wedge \psi$
chp stp:	$(\tau \wedge \circ\, \varphi); \psi$	$\leftrightarrow$	$\tau \wedge \circ\, (\varphi; \psi)$

Table 7: Rules for executing sequential composition

4.4 Executing sequential composition

The execution of sequential composition of programs $\varphi; \psi$ is more complicated as we cannot give a simple equivalence which rewrites a composition to normal form. The problem is that the first formula φ could take an unknown number of steps to execute. Only after φ has terminated, we continue with executing ψ.

Rules for the execution of $\varphi; \psi$ are given in Table 7. In order to execute composition, the idea is to first rewrite formula φ to normal form

$$\left(\tau_0 \wedge \mathbf{last} \vee \bigvee (\exists\, a_i.\ \tau_i \wedge \circ\, \varphi_i)\right); \psi$$

The first sub-formula φ can be rewritten with rule *chp lem* of Table 7. If it is valid that φ_1 implies φ_2, then $\varphi_1; \psi$ also implies $\varphi_2; \psi$. (This rule is a so-called congruence rule.) After rewriting the first sub-formula to normal form, we rewrite the composition operator. According to rules *chp dis* and *chp ex*, composition distributes over disjunction and existential quantification. If we apply these rules to the formula above, we receive a number of cases

$$(\tau_0 \wedge \mathbf{last}); \psi \vee \bigvee \exists\, a_{i,0}.\ (\tau_{i,0} \wedge \circ\, \varphi_{i,0}); \psi$$

In the first case, program φ terminates, in the other cases, the program takes a step τ_i and continues with program ψ_i. Rules *chp lst* and *chp stp* can be used to further rewrite the composition. Dynamic variables A stutter in the last step, and therefore the primed and double primed variables A' and A'' of τ are replaced by the corresponding unprimed variables if the first sub-formula terminates. The two rules give

$$\tau_0{}^{A,A}_{A',A''} \wedge \psi \vee \bigvee \exists\, a_{i,0}.\ \tau_{i,0} \wedge \circ\, (\varphi_{i,0}; \psi)$$

In the first case, φ has terminated and we still need to execute ψ to arrive with a formula in normal form. In the other cases, we have successfully separated the formula into the first transition $\tau_{i,0}$ and the corresponding rest of the program $\varphi_{i,0}; \psi$.

4.5 Interleaving

As with sequential composition, the interleaving of programs cannot be executed directly, but the sub-formulas need to be rewritten to normal form first. The

basic operator for interleaving is the left interleaving operator $\varphi \,||^< \psi$ which gives precedence to the left process. In order to execute $||^<$, the first sub-formula must be rewritten to normal form before the operator itself can be rewritten.

$$\frac{\vdash \varphi_1 \rightarrow \varphi_2}{\vdash \varphi_1 \,||^< \psi \rightarrow \varphi_2 \,||^< \psi} \; \textit{ilvl lem}$$

$$\begin{array}{lll}
\textit{ilvl dis:} & (\varphi_1 \vee \varphi_2) \,||^< \psi & \leftrightarrow \quad \varphi_1 \,||^< \psi \vee \varphi_2 \,||^< \psi \\
\textit{ilvl ex:} & (\exists\, a.\ \varphi) \,||^< \psi & \leftrightarrow \quad \exists\, a_0.\ \varphi_a^{a_0} \,||^< \psi \\
\multicolumn{3}{l}{\quad a_0 \text{ fresh with respect to } (\text{free}(\varphi) \setminus \{a\}) \cup \text{free}(\psi)} \\
\textit{ilvl lst:} & (\tau \wedge \mathbf{last}) \,||^< \psi & \leftrightarrow \quad \tau^{A,A}_{A',A''} \wedge \psi \\
\textit{ilvl stp:} & (\tau \wedge \neg\, \mathbf{blocked} \wedge \circ\, \varphi) \,||^< \psi & \\
 & \leftrightarrow \exists\, a_2.\ (\tau^{a_2}_{A''} \wedge \neg\, \mathbf{blocked} \wedge \circ\, ((A = a_2 \wedge \varphi) \,||\, \psi)) & \\
\textit{ilvl blk:} & (\tau \wedge \mathbf{blocked} \wedge \circ\, \varphi) \,||^< \psi & \\
 & \leftrightarrow \exists\, a_{2,1}.\ (\exists\, a_1.\ \tau^{a_1,a_{2,1}}_{A',A''}) \wedge (A = a_{2,1} \wedge \varphi) \,||^<_b \psi &
\end{array}$$

Table 9: Rules for executing left interleaving

For left interleaving, the rules of Table 9 are similar to the rules for sequential composition. Congruence rule *ilvl lem* makes it possible to rewrite the first sub-formula to normal form. Similar to chop, left interleaving also distributes over disjunction (*ilvl dis*) and existential quantifiers (*ilvl ex*). If the first formula terminates, execution continues with the second (*ilvl lst*). This is similar to rule *chp lst*. Otherwise, execution depends on the first process being blocked. If it is not blocked, rule *ilvl stp* executes the transition and continues with interleaving the remaining φ with ψ. Note that the double primed variables of τ are replaced by static variables a_2 which must be equal to the unprimed variables the next time a transition of the first process is executed. This is to establish the environment transition of the first process as a relation which also includes transitions of the second. If the first process is blocked, then rule *ilvl blk* executes the blocked transition; the process actively waits while being blocked. However, the primed variables of τ are replaced by static variables a_1: the blocked transition of the first process does not contribute to the transition of the overall interleaving. Instead, a transition of the other process is executed.

The situation where the first process is blocked and it remains to execute a transition of the second process is represented by a derived operator $\varphi \,||^<_b \psi$ which is defined

$$\varphi \,||^<_b \psi \quad :\equiv \quad (\mathbf{blocked} \wedge \circ\, \varphi) \,||^< \psi$$

and for which the rules of Table 11 are applicable. Again, the rules are very similar to the rules above. A congruence rule *ilvlb lem* ensures that the second sub-formula can be rewritten to normal form. With rules *ilvlb dis* and *ilvlb ex*, the operator distributes over disjunction and existential quantification. If the second process terminates, *ilvlb lst* is applicable, otherwise *ilvlb stp* can be applied.

Similar rules have been defined for all the operators of our logic [2]. In total,

$$\frac{\vdash \varphi_1 \to \varphi_2}{\vdash \psi \,||_b^< \, \varphi_1 \to \psi \,||_b^< \, \varphi_2} \; \textit{ilvlb lem}$$

ilvlb dis:	$\psi \,\|\|_b^< \, (\varphi_1 \vee \varphi_2)$	$\leftrightarrow$	$\psi \,\|\|_b^< \, \varphi_1 \vee \psi \,\|\|_b^< \, \varphi_2$
ilvlb ex:	$\psi \,\|\|_b^< \, (\exists\, a.\, \varphi)$	$\leftrightarrow$	$\exists\, a_0.\, \psi \,\|\|_b^< \, \varphi_a^{a_0}$
	a_0 fresh with respect to $(\mathrm{free}(\varphi) \setminus \{a\}) \cup \mathrm{free}(\psi)$		
ilvlb lst:	$\psi \,\|\|_b^< \, (\tau \wedge \mathbf{last})$	$\leftrightarrow$	false
ilvlb stp:	$\psi \,\|\|_b^< \, (\tau \wedge \circ\, \varphi)$	$\leftrightarrow$	$\exists\, a_{2,2}.\, \tau_{A''}^{a_{2,2}} \wedge \circ\, (\psi \,\|\|\, (A = a_{2,2} \wedge \varphi))$

Table 11: Rules for executing blocked left interleaving

every temporal formula can be rewritten to normal form, an overall step can be executed, and the process of symbolic execution can be repeated over again.

4.6 Sequencing

The execution of interleaved parallel processes is nondeterministic. As a consequence, a large number of different paths of execution have to be considered. Very often, however, the order of execution has no effect on the computation; following different paths leads to the same state. To minimize the number of paths, the granularity of a system step can be redefined: two statements of a process can be executed in a single step, if they do not affect the global state; if a process only assigns local variables, then the global state is not changed, and only the stuttering behavior of the process differs if statements are combined. However, this approach depends on the type of property to verify; the property must be invariant to stuttering steps. Therefore, we have taken a different approach.

Our approach is simple but effective. Initially, all paths of execution are considered. However, if the same system configuration results from executing a number of steps but in a different order, then the two resulting premises are combined. We do not distinguish between program and property and only, if the program configuration and the configuration of the temporal formulas are equal, the premises are combined. As a consequence, our approach can be applied to any type of temporal property.

Two premises are combined, if the system is residing in the same configuration. A system configuration is defined by the temporal formulas of a premise. In order to automate sequencing, we have defined a syntactic criterion to determine the current system configuration. For this purpose, function $\mathrm{conf}(\varphi)$ extracts a set of temporal formulas from a given sequent $\Gamma \vdash \Delta$ (see Table 12). Condition $\mathrm{conf}(\Gamma_1 \vdash \Delta_1) = \mathrm{conf}(\Gamma_2 \vdash \Delta_2)$ is a (computable) indication to combine two premises. In order to combine two premises, we have generalized the concept of a sequent rule to allow for several conclusions within a single rule. Our sequencing rule reads:

$$\frac{\vdash (\bigwedge \Gamma \to \bigvee \Delta_1) \wedge (\bigwedge \Gamma_2 \to \bigvee \Delta_2)}{\Gamma_1 \vdash \Delta_1 \quad \Gamma_2 \vdash \Delta_2} \; \textit{seq,} \quad \textit{if } \mathrm{conf}(\Gamma_1 \vdash \Delta_1) = \mathrm{conf}(\Gamma_2 \vdash \Delta_2)$$

$$\begin{array}{rcl}
\mathrm{conf}(\Gamma \vdash \Delta) & :\equiv & \bigcup_{\varphi\in\Gamma} \mathrm{conf}^-(\varphi) \cup \bigcup_{\varphi\in\Delta} \mathrm{conf}(\varphi) \\
\\
\mathrm{conf}(\varphi_{\mathbf{PL}}) & :\equiv & \emptyset \\
\mathrm{conf}(\neg\, \varphi) & :\equiv & \{\mathrm{neg}(\psi) \mid \psi \in \mathrm{conf}^-(\varphi)\} \\
\mathrm{conf}(\varphi \vee \psi) & :\equiv & \mathrm{conf}(\varphi) \cup \mathrm{conf}(\psi) \\
\mathrm{conf}(\varphi \rightarrow \psi) & :\equiv & \mathrm{conf}(\neg\, \varphi) \cup \mathrm{conf}(\psi) \\
\mathrm{conf}(\varphi) & :\equiv & \{\varphi\} \\
\\
\mathrm{conf}^-(\varphi_{\mathbf{PL}}) & :\equiv & \emptyset \\
\mathrm{conf}^-(\neg\, \varphi) & :\equiv & \{\mathrm{neg}(\psi) \mid \psi \in \mathrm{conf}(\varphi)\} \\
\mathrm{conf}^-(\varphi \wedge \psi) & :\equiv & \mathrm{conf}^-(\varphi) \cup \mathrm{conf}^-(\psi) \\
\mathrm{conf}^-(\varphi) & :\equiv & \{\varphi\} \\
\\
\mathrm{neg}(\neg\, \varphi) & :\equiv & \varphi \\
\mathrm{neg}(\varphi) & :\equiv & \neg\, \varphi
\end{array}$$

Table 12: Definition of a system configuration $\mathrm{conf}(\Gamma \vdash \Delta)$

Only if the system configuration of the two conclusions are equal, the premises are combined. In this case, the temporal formulas of both premises are equivalent, and only the first order formulas differ. Thus, the conjunction of both premises is subject to significant simplification.

4.7 Induction

If execution loops, the proof requires an inductive argument. Our induction method is based on Noetherian induction. Given a well-founded ordering $<$ and an expression e, a general rule for induction is as follows:

$$\frac{e = a, \bullet\, \Box\, (e < a \rightarrow \mathbf{ih}), \Gamma \vdash \Delta}{\Gamma \vdash \Delta}\ \mathrm{ind}(e)$$

where $\mathbf{ih} :\equiv \bigwedge \Gamma \rightarrow \bigvee \Delta$ and a is fresh with respect to $\mathrm{free}(e, \Gamma, \Delta)$. For any proof obligation $\Gamma \vdash \Delta$, rule *ind* can be applied to receive an induction hypothesis as an additional precondition: starting with the next state, the hypothesis can always be applied, if e has decreased, static variable a containing the original value of e.

It is not necessary to provide an ordering in all cases. If a liveness property $\Diamond\, \varphi$ is known, it is possible to induce over the number of steps it takes to reach the first state satisfying φ. A derived rule is as follows:

$$\frac{\bullet\ \mathbf{ih\ until}\ \varphi, \Gamma \vdash \Delta}{\Diamond\, \varphi, \Gamma \vdash \Delta}\ \text{ind ev}$$

Liveness properties can be derived from other temporal operators as well.

$$\Box\, \varphi \leftrightarrow \neg\, \Diamond\, \neg\, \varphi$$

Thus, a safety property in the succedent can be turned into a liveness property in the antecedent and induction is possible.

Our strategy of symbolic execution ensures that after a finite number of steps, execution terminates or the system loops and induction can be applied. However, it might be necessary to generalize the first order formulas with an invariant. It is an important principle of our induction method that only first order formulas need to be generalized.

5 Implementation

The interactive proof method has been implemented in KIV [3], an interactive theorem prover which is available at [10]. KIV supports algebraic specifications, predicate logic, dynamic logic, and higher order logic. Especially, reasoning in predicate logic and dynamic logic is very elaborate. Support for concurrent systems and temporal logic has been added. Considerable effort has been spent to ensure that the calculus rules are automatically applied to a large extent. Almost all of the rules are invertible ensuring that, if the conclusion is provable, the resulting premises remain valid. The overall strategy is

- to symbolically execute a given proof obligation,
- to simplify the PL formulas describing the current state,
- to combine premises with the same system configuration, and
- to use induction, if a system loop has been executed.

This strategy is illustrated with an example proof in the following section.

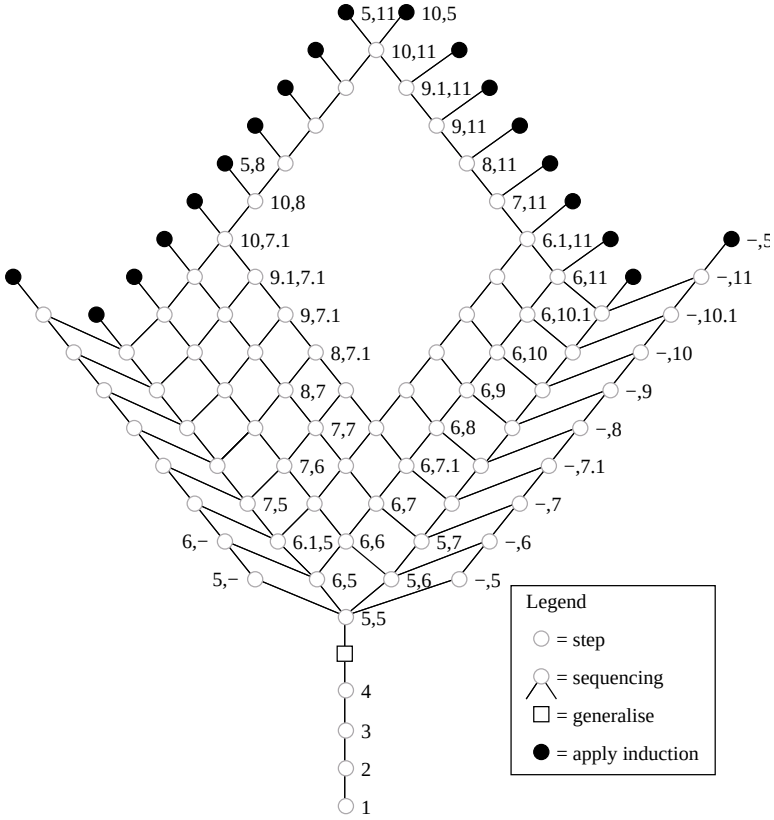


Figure 3: Proof for property (1)

6 Example proofs

6.1 Verifying liveness

In order to verify the liveness property (1) for the algorithm of Fig. 2, we apply the strategy of symbolic execution to receive the proof graph of Figure 3.

The different nodes of the proof graph are annotated with numbers to refer to the program positions of the algorithm in Figure 2. Initially, the program is at position 1 (bottom node). Executing the first step leads to position 2 and so on. After the variables have been initialized, two processes are spawned and consequently the following nodes are annotated with two numbers. The two processes start out at positions $5, 5$. In the following step, either a transition of the first or the second process is executed. The while loops of both processes either terminate or loop. Therefore, step execution gives four successor states $5, -$ (the second process has terminated), $6, 5$ (the first process loops), $5, 6$ (the second process loops), and $-, 5$ (the first process has terminated). Continuing with executing the algorithm in the four states again results in one to three successor states each; in total, we receive eight states. With sequencing, the number of states can be reduced to five different states. The process of step execution and sequencing is repeated until program positions are reached which

have been encountered before. In this case, induction can be applied.

The resulting proof represents all possible execution paths. In Figure 3, a selection of paths have been annotated with corresponding program positions. (Note: if a sub procedure is executed, the corresponding program position $a.b$ consists of position a of the calling procedure followed by position b of the sub procedure.) The empty space in the middle of the graph represents unreachable program positions; because execution is synchronized with a semaphore it is impossible for both processes to reside in their critical section at the same time.

The values of variables are represented by first order formulas. For executing loops, these formulas must be generalized to be invariant. At program position 4, the variable settings read

$$k \leq n \wedge S = 1 \wedge Y_1 = n \wedge Y_2 = 1 \wedge B = 1 \ .$$

For verifying the liveness property, the values of the two counters Y_1 and Y_2 are generalized appropriately:

$$k \leq n \wedge S = 1 \wedge n - k \leq Y_1 \leq n \wedge 1 \leq Y_2 \leq k + 1 \ .$$

Knowledge about variable B can be discarded. In addition to the invariant, a Noetherian ordering must be provided to ensure that the algorithm loops only finitely often. In our case, the function

$$(Y_1 - (n - k)) + ((k + 1) - Y_2)$$

is always greater or equal to zero and decreases after each loop execution.

Finding an invariant and providing an ordering are the only interactive steps in our example. Step execution, sequencing, and application of the induction hypothesis are fully automatic.

6.2 Verifying safety

The proof for property (2) is almost identical to the proof in Figure 3. The same steps are executed and the same proof graph results. The only differences are a more complex invariant, no necessity to provide a Noetherian ordering, and more complex first order reasoning.

For the invariant, it is important to properly generalism B. Our invariant for property (2) reads

$$\begin{aligned} &k \leq n \wedge S = 1 \wedge n - k \leq Y_1 \leq n \wedge 1 \leq Y_2 \leq k + 1 \\ &\wedge Y_1 + Y_2 \leq n + 1 \wedge B = \left(\textstyle\prod_{i=Y_1+1}^{n} i\right) / \left(\textstyle\prod_{i=1}^{Y_2-1} i\right) \ . \end{aligned}$$

It is not necessary to provide an ordering for induction because of the following reason. In property (2), the formula on the right hand side of the implication is preceeded by an always operator $\Box\ \varphi$ which is equivalent to a negated eventually operator $\neg \Diamond \neg \varphi$. Thus, $\Diamond \neg \varphi$ can be assumed: eventually (after a finite number of steps) $\neg \varphi$ holds. Induction is over the number of steps it takes to reach a state where φ is violated.

After each step execution, the first order formulas relate the values of variables before and after the transition. This relation must be simplified as far as possible. For the simple invariant of the liveness property, simplification is fully automatic. The more complex invariant of the safety property requires additional first order reasoning. For example, at program position $7, 7$ the first order formula reads

$$k \leq n \wedge S = 0 \wedge Y_1 \neq 0 \wedge n < Y_1 + k \wedge Y_1 \leq n \wedge Y_2 \neq 0 \wedge Y_2 < k \wedge Y_2 < n \\ \wedge\, Y_1 + Y_2 \leq n \wedge Y_2 + 1 < Y_1 + k \wedge B = \left(\prod_{i=Y_1+1}^{n} i\right) / \left(\prod_{i=1}^{Y_2-1} i\right)$$

This formula slightly differs from the original invariant. Semaphore S is zero and additional knowledge $Y_1 \neq 0$ etc. has been derived. Furthermore, rules have been applied to eliminate subtraction $n - k$. Immediately after the transition to program position $8, 7$, we receive

$$\textit{/* new static variables store old value */} \\ k \leq n \wedge s = 0 \wedge y_1 \neq 0 \wedge n < y_1 + k \wedge y_1 \leq n \wedge y_2 \neq 0 \wedge y_2 < k \wedge y_2 < n \\ \wedge\, y_1 + y_2 \leq n \wedge y_2 + 1 < y_1 + k \wedge b = \left(\prod_{i=y_1+1}^{n} i\right) / \left(\prod_{i=1}^{y_2-1} i\right) \\ \textit{/* assignment turned into a relation */} \\ \wedge\, T = b * Y_1 \wedge B = b \wedge Y_1 = y_1 \wedge Y_2 = y_2 \wedge S = s$$

The assignment $T := B * Y_1$ has been executed. New static variables b, y_1, y_2, s have been introduced to store the original values of the dynamic variables, and the assignment has been turned into a relation between these static variables and the dynamic variables in the next state. With equational reasoning, the newly introduced static variables are automatically eliminated to receive

$$k \leq n \wedge S = 0 \wedge Y_1 \neq 0 \wedge n < Y_1 + k \wedge Y_1 \leq n \wedge Y_2 \neq 0 \wedge Y_2 < k \wedge Y_2 < n \\ \wedge\, Y_1 + Y_2 \leq n \wedge Y_2 + 1 < Y_1 + k,\ \wedge\, B = \left(\prod_{i=Y_1+1}^{n} i\right) / \left(\prod_{i=1}^{Y_2-1} i\right) \\ \wedge\, T = \left[\left(\prod_{i=Y_1+1}^{n} i\right) / \left(\prod_{i=1}^{Y_2-1} i\right)\right] * Y_1$$

Additional first order reasoning is required to simplify the value of T. Functions $\prod$ and / satisfy the following lemma:

$$Y_1 + Y_2 \leq n, Y_1 \leq n \vdash \left[\left(\prod_{i=Y_1+1}^{n} i\right) / \left(\prod_{i=1}^{Y_2-1} i\right)\right] * Y_1 = \left(\prod_{i=Y_1}^{n} i\right) / \left(\prod_{i=1}^{Y_2-1} i\right) .$$

If the two preconditions are satisfied, then the left hand side of the equation can be replaced by the right hand side. With this lemma, the value of T can be simplified. A number of lemmas have been added to the library of simplifier rules to complete the proof. The complexity of the lemmas vary from simple lemmas, e.g. $\prod_{i=1}^{0} i = 1$, to more complex lemmas as above.

This second proof is a good example for a combination of complex first order reasoning and reasoning in temporal logic. Again, the temporal logic steps are fully automatic and it remains to supply an invariant and additional lemmas to simplify the first order formulas.

6.3 Verification in KIV

Figure 4 contains a screen-shot to illustrate how to construct proofs in KIV. In the lower right, the proof graph is displayed. The visible portion of the graph

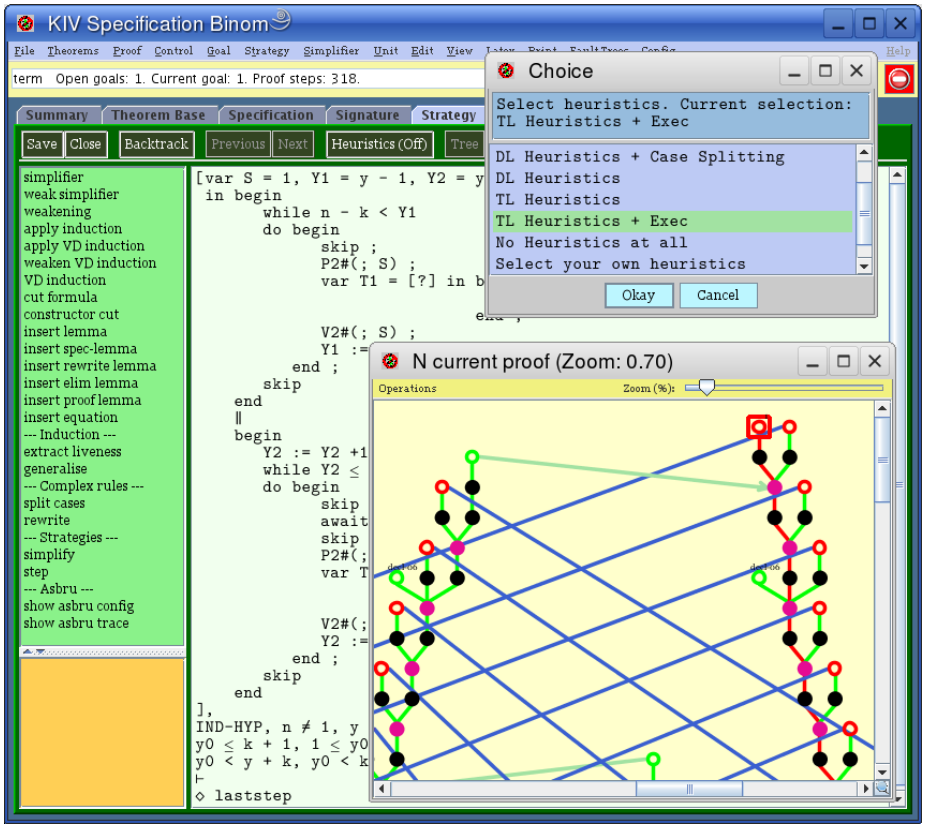


Figure 4: Verification of Binom example in KIV

corresponds to the upper part of Figure 3. The main area in the large window contains the sequent under verification with the partially executed parallel program and the first order formulas describing the current state. A list of applicable proof rules is displayed to the left. Heuristics are used to automatically apply these rules. Two set of heuristics `TL Heuristics` and `TL Heuristics + Exec` implement the overall strategy of Section 5.

7 Conclusion

We have successfully carried over the strategy of symbolic execution to verify arbitrary temporal properties for concurrent systems. The proof method is based on symbolic execution, sequencing, and induction. How to symbolically execute arbitrary temporal formulas – parallel programs being just a special case thereof – has been explained in this paper.

Our proof method is easily extendable to other temporal operators. For every operator, a set of rules must be provided to rewrite the operator to normal form. With rules similar to the ones of Tables 7 and 9, we support in KIV operators for Dijkstra's choice, synchronous parallel execution, and interrupts. Furthermore, we have integrated STATEMATE state charts [16] as well as UML state charts [4] as alternative formalisms to define concurrent systems. For all of our extensions, the strategy of sequencing and induction has remained unchanged and arbitrary temporal formulas can be verified.

Using double primed variables, we have defined a compositional semantics for every operator including interleaving of formulas. With compositional operators, proofs can be decomposed. This is very important to verify large case studies. How to decompose proofs in our logic is subject to further investigations.

The implementation in KIV has shown that symbolic execution can be automated to a large extent. We have applied the strategy to small and medium size case studies. Currently, the strategy is applied in a European project called Protocure to verify medical guidelines which can be seen as yet another form of concurrent system. Overall, we believe that the strategy has the potential to make interactive proofs in (linear) temporal logic in general more intuitive and automatic.

References

[1] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.

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

[3] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*. Springer LNCS 1783, 2000.

[4] S. Bäumler, M. Balser, A. Knapp, W. Reif, and A. Thums. Interactive verification of uml state machines. In *Formal Methods and Software Engineering*, number 3308 in LNCS. Springer, 2004.

[5] N. Bjørner, A. Brown, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma, and T. Uribe. Verifying temporal properties of reactive systems: A step tutorial. In *Formal Methods in System Design*, pages 227–270, 2000.

[6] D. Harel and D. Peleg. Process logic with regular formulas. *Theoretical Computer Science*, 38:307–322, 1985.

[7] David Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 2, pages 496–604. Reidel, 1984.

[8] David Harel and Eli Singerman. Computation path logic: An expressive, yet elementary, process logic. *Annals of Pure and Applied Logic*, pages 167–186, 1999.

[9] M. Heisel, W. Reif, and W. Stephan. A Dynamic Logic for Program Verification. In A. Meyer and M. Taitslin, editors, *Logical Foundations of Computer Science*, LNCS 363, pages 134–145, Berlin, 1989. Logic at Botik, Pereslavl-Zalessky, Russia, Springer.

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

[11] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[12] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems—Safety*. Springer, 1995.

[13] Ben Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.

[14] D. Peleg. Concurrent dynamic logic. *J. ACM*, 34(2):450–479, 1987.

[15] R. Sherman, A. Pnueli, and D. Harel. Is the interesting part of process logic uninteresting?: a translation from pl to pdl. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 347–360, New York, NY, USA, 1982. ACM Press.

[16] Andreas Thums. *Formale Fehlerbaumanalyse*. PhD thesis, Universität Augsburg, Augsburg, Germany, 2004. (in German).