Verifying Concurrent Systems with Symbolic Execution

Michael Balser   Christoph Duelli   Wolfgang Reif
Gerhard Schellhorn

March 25, 2003

Lehrstuhl Softwaretechnik und Programmiersprachen
Universitt Augsburg, D-86135 Augsburg, Germany
email: {balser,duelli,reif,schellhorn}@informatik.uni-augsburg.de

Abstract

Current techniques for interactively proving temporal properties of concurrent systems translate transition systems into temporal formulas by introducing program counter variables. Proofs are not intuitive, because control flow is not explicitly considered. For sequential programs symbolic execution is a very intuitive, interactive proof strategy. In this paper we will adopt this technique for parallel programs. Properties are formulated in interval temporal logic. An inplementation in the interactive theorem prover KIV has shown that this technique offers a high degree of automation and allows simple, local invariants.

1 Introduction

As an example of a concurrent system, consider the parallel program *Binom* which is shown in Fig. 1 and has been taken from [15]. Two parallel processes calculate the binomial coefficient $\binom{n}{k}$. Because of

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

the first process multiplies all values $n, n-1, \ldots, n-k+1$ and the second process divides the intermediate result b by $1, 2, \ldots, k$. Awaiting $y_1 + y_2 \leq n$ guarantees that the division is without remainder.

Auxiliary variables t_1 and t_2 are used to simulate fine granular read and write access to the shared variable b and a semaphore s is used to synchronize processes in accessing b. Labels $l, l_1, \ldots$ and $q, q_1, \ldots$ were added to the program to refer to specific locations. After execution, variable b should contain the result. Therefore, an interesting property to prove is that finally

$$b = \binom{n}{k}.$$

1

$$y_1 := n;\ y_2 := 1;\ b := 1;\ s := 1;$$

l: **while** $y_1 > n - k$ **do** $\quad\|\quad$ q: **while** $y_2 \leq k$ **do**

$\qquad$ $l_1 : P(s);$ $\qquad\qquad\qquad\quad$ $q_1 : $ **await** $y_1 + y_2 \leq n$ **do skip**

$\qquad$ $l_2 : t_1 := b * y_1;$ $\qquad\qquad\quad$ $q_2 : P(s);$

$\qquad$ $l_3 : b := t_1;$ $\qquad\qquad\qquad\quad$ $q_3 : t_2 := b$ **div** $y_2;$

$\qquad$ $l_4 : V(s);$ $\qquad\qquad\qquad\quad$ $q_4 : b := t_2;$

$\qquad$ $l_5 : y_1 := y_1 - 1$ $\qquad\qquad\;$ $q_5 : V(s);$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $q_6 : y_2 := y_2 + 1$

where $P(r) \equiv$ **await** $s > 0$ **do** $s := s - 1,$

$\qquad$ $V(r) \equiv s := s + 1,$

$\qquad$ **skip** $\equiv x := x$ for an arbitrary x.

Figure 1: Program *Binom*

For this, an important temporal property is that exclusive access to the critical section guarded by semaphore s is always guaranteed

$$\Box\,(\neg\,(l_2 \vee l_3 \vee l_4) \vee \neg\,(q_3 \vee q_4 \vee q_5)).$$

While the first property argues about the final state of execution, the second property is concerned with states during execution.

The parallel program *Binom* defines a transition system for which some temporal properties have to be proven. In special cases, e.g. if the transition system is finite, model checking [6] can be used. For infinite systems – like in the example – proofs often cannot be found automatically and an interactive proof technique is needed. For an interactive strategy it is very important that proofs are easy to understand, but also as automatic as possible. Current techniques for interactively verifying temporal properties do not explicitly consider the control flow of the transition system. In TLA [14] the system is transformed into a temporal formula by introducing program counter variables for encoding control flow. [15] generates an unstructured set of transitions where each transition is described in predicate logic. In both cases the control flow is hidden. As a consequence the interactive proof does not follow the control flow of the original transition system and proofs lack intuition. In this paper we propose a proof method which explicitly considers the control flow of the transition system and allows to step through the system with symbolic execution. Symbolic execution is an intuitive proof method based on computing strongest post conditions which has already been successfully applied to the interactive verification of sequential programs [7]. Furthermore, it can be automated to a large extent.

Using symbolic execution with induction for the verification of parallel programs has been applied before. [17] uses Hoare logic to verify properties of a subset of linear temporal logic, [1] verifies relations between input and output of parallel programs by symbolically executing those. In the long run, our goal is to construct a proof method for verifying arbitrary temporal properties for a variety of concurrent system descriptions with an operational semantics. In this paper we are restricted to parallel programs as system description and only consider inductive proofs of simple safety properties. However we show how to execute arbitrary formulas of Interval Temporal

Logic and we have evidence that the induction technique can also be generalised to a larger set of temporal formulas.

This paper concentrates on the underlying calculus of the proof method. In order to apply this interactive method in practice, topics such as comprehensibility (possibly through graphical visualization similar to STeP [4]), and tool support also have to be considered. This is not done here. However, it should be noted that the calculus has been implemented in KIV [2], an integrated development environment for formally developing correct systems. Already, KIV offered strong proof support for algebraic specifications [9] with higher order logic, and dynamic logic [10] for the verification of sequential programs. Proof support for temporal logic and parallel programs has been added.

Section 2 introduces syntax and semantics of the underlying logic. In section 3 the proof method of symbolically executing concurrent systems is introduced and also an invariant technique for verifying loops in the control flow is provided. A comparison to other techniques is given in section 4. Section 5 concludes.

2 Semantics

In this section we will define the syntax and semantics of the interval temporal logic our proof method is based on. Parallel programs are used to describe possible sequences of execution states.

We assume the reader is familiar with the notions of first order logic: a many-sorted first order signature SIG and a familiy of variables X are used to define first order terms τ, boolean expressions ε and formulas p over SIG and X. Given a fixed algebra $\mathcal{A}$ for SIG and a valuation σ (a mapping from X to the carriers of $\mathcal{A}$, also called a state) we can define the semantics $[\![\tau]\!]_\sigma$ of terms and $[\![p]\!]_\sigma$ of formulas (the latter being a boolean value).

The programs we use consist of assignments ($x := \tau$), compounds ($\alpha; \beta$), conditionals (**if** ε **then** α **else** β), while loop (**while** ε **do** α), interleaved parallel execution ($\alpha \parallel \beta$), and await statement (**await** ε **do** $x := \tau$)[1]. For technical reasons we also use the empty program **E** which does nothing. We define (**E**; α), ($\alpha;$ **E**), (**E** $\parallel \alpha$), and ($\alpha \parallel$ **E**) all to be equal to α. Therefore we never use the empty program in compounds or interleaved programs.

To define the operational semantics of programs, configurations $\langle \alpha_i, \sigma_i \rangle$ are used, which consist of a program α_i to be executed and a current state σ_i of the variables. A relation '$\longrightarrow$' (written infix) is inductively defined by rules to describe possible successor configurations, when executing one program step. Some example rules are given in Fig. 2. The definition is taken from [1], where a full list of rules can be found.

The rules in the upper row define the semantics of conditionals and loops, when the test is positive. The first rule in the middle row gives the effect of an assignment (alteration of the value of x). The semantics of the await statement is given by the right rule of the middle row: execution proceeds only when the test is true. Finally the last row of Fig. 2 gives the two possible next steps of interleaved execution: either a step of α or one of β is executed.

---

[1]For simplicity of presentation, we omit the general statement **await** ε **do** α from [1], where α is executed in one step (blocked).

$$\frac{[\![\varepsilon]\!]_\sigma = \text{true}}{\langle \textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta, \sigma \rangle \longrightarrow \langle \alpha, \sigma \rangle} \qquad \frac{[\![\varepsilon]\!]_\sigma = \text{true}}{\langle \textbf{while } \varepsilon \textbf{ do } \alpha, \sigma \rangle \longrightarrow \langle \alpha; \textbf{while } \varepsilon \textbf{ do } \alpha, \sigma \rangle}$$

$$\frac{}{\langle x := \tau, \sigma \rangle \longrightarrow \langle \textbf{E}, \sigma[x \leftarrow [\![\tau]\!]_\sigma] \rangle} \qquad \frac{[\![\varepsilon]\!]_\sigma = \text{true}}{\langle \textbf{await } \varepsilon \textbf{ do } x := \tau, \sigma \rangle \longrightarrow \langle \textbf{E}, \sigma[x \leftarrow [\![\tau]\!]_\sigma] \rangle}$$

$$\frac{\langle \alpha, \sigma \rangle \longrightarrow \langle \alpha', \sigma' \rangle}{\langle \alpha \parallel \beta, \sigma \rangle \longrightarrow \langle \alpha' \parallel \beta, \sigma' \rangle} \qquad \frac{\langle \beta, \sigma \rangle \longrightarrow \langle \beta', \sigma' \rangle}{\langle \alpha \parallel \beta, \sigma \rangle \longrightarrow \langle \alpha \parallel \beta', \sigma' \rangle}$$

Figure 2: Operational semantics of programs

Based on the relation $\longrightarrow$ the semantics $[\![\alpha]\!]_\sigma$ of a program α when started in a state σ is defined to be the set of all (finite or infinite) sequences of configurations $(\langle \alpha_0, \sigma_0 \rangle, \langle \alpha_1, \sigma_1 \rangle, \ldots)$ which start with $\langle \alpha, \sigma \rangle$, end with the empty program $\textbf{E}$ when finite, and which satisfy $\langle \alpha_i, \sigma_i \rangle \longrightarrow \langle \alpha_{i+1}, \sigma_{i+1} \rangle$ for all suitable i.

In proofs we will also use labelled programs $l : \alpha$. We require that labels are taken from a special subset $\mathbb{L}$ of the boolean variables that may not be used in programs other than as labels. The semantics of labelled programs could be given using similar, but technically more complex rules than Fig. 2. We prefer to give an informal definition. The semantics of a labelled program is the same set of configuration sequences as the corresponding unlabelled one, except that for each configuration (α, σ) the semantics of labels is modified such that $\sigma(l)$ is true for all labels $l \in lab(\alpha)$ and false otherwise. Function lab, which yields the labels at the beginning of α, is defined to be $\{l\} \cup lab(\alpha)$ for $l : \alpha$. $lab(\alpha \parallel \beta)$ is $lab(\alpha) \cup lab(\beta)$, and $lab(\alpha; \beta)$ is $lab(\alpha)$. For all other programs α, $lab(\alpha)$ is the empty set.

Based on the semantics of programs and first order formulas we can now define the semantics of the formulas of interval temporal logic (ITL) [16] in Fig. 2. The definition uses (finite and infinite) sequences of states $\boldsymbol{\sigma} = (\sigma_0, \sigma_1, \ldots)$ called intervals. $\boldsymbol{\sigma}_i$ abbreviates $(\sigma_i, \sigma_{i+1}, \ldots)$. Other temporal operators can be derived: weak next $\bigcirc \varphi \equiv \neg (\textbf{unit} \frown \neg \varphi)$, $\textbf{last} \equiv \bigcirc \text{false}$, $\Box \varphi \equiv \varphi \textbf{ unless } \text{false}$, and $\Diamond \varphi \equiv \neg \Box \neg \varphi$. Because programs are formulas we can write $\alpha \rightarrow \varphi$ to express that all execution traces of α have property φ.

In ITL, formula $\Box \text{Prop}$, where $\text{Prop} \equiv \textbf{last} \rightarrow b = \binom{n}{k}$, expresses that $b = \binom{n}{k}$ should hold in the last state of an interval. We use a sequent calculus to prove this property for program $Binom$ (see Fig. 1), the desired sequent reads as follows

$$0 \le k \le n, (y_1 := n; y_2 := 1; \ldots) \vdash \Box \text{Prop}.$$

---

[2]Star formulas φ^*, which are defined in ITL, were omitted for simplicity. The operator $\varphi \textbf{ unless } \psi$ has been added as basic operator, since it cannot be derived.

$$\begin{array}{lll}
\boldsymbol{\sigma} \models p & \text{iff} & [\![p]\!]_{\sigma_0} = \text{true} \\
\boldsymbol{\sigma} \models \varphi \wedge \psi & \text{iff} & \boldsymbol{\sigma} \models \varphi \text{ and } \boldsymbol{\sigma} \models \psi \\
\boldsymbol{\sigma} \models \neg\,\varphi & \text{iff} & \text{not } \boldsymbol{\sigma} \models \varphi \\
\boldsymbol{\sigma} \models \textbf{unit} & \text{iff} & \boldsymbol{\sigma} = (\sigma_0, \sigma_1) \\
\boldsymbol{\sigma} \models \alpha & \text{iff} & \text{there is a run } ((\alpha_0, \sigma_0), (\alpha_1, \sigma_1), \ldots) \text{ in } [\![\alpha]\!]_{\sigma_0}
\end{array}$$

$$\boldsymbol{\sigma} \models \varphi \textbf{ unless } \psi \quad \text{iff} \quad \begin{cases} \text{either for all } i \; \boldsymbol{\sigma}_i \models \varphi \\ \text{or there is an } i \text{ with } \boldsymbol{\sigma}_j \models \varphi \text{ for all } j < i \text{ and } \boldsymbol{\sigma}_i \models \psi \end{cases}$$

$$\boldsymbol{\sigma} \models \varphi \frown \psi \quad \text{iff} \quad \begin{cases} \text{either } \boldsymbol{\sigma} \text{ is infinite and } \boldsymbol{\sigma} \models \varphi \\ \text{or there is an } i \text{ such that } (\sigma_0, \ldots \sigma_i) \models \varphi \text{ and } \boldsymbol{\sigma}_i \models \psi \end{cases}$$

Figure 3: Semantics of interval temporal logic[2]

3 Proof Method

3.1 Symbolic Execution

For symbolic execution the following rule is used

$$\frac{\mathcal{N}_m(\Gamma) \vdash \mathcal{N}_m(\Delta) \quad \textbf{last}, \mathcal{L}(\Gamma) \vdash \mathcal{L}(\Delta)}{\Gamma \vdash \Delta} \; \text{step.}$$

Two cases have to be treated as premises. Either a step is taken or – for finite intervals – the last step has been reached.

In the first case the originally considered interval $(\sigma_0, \sigma_1, \ldots)$ is reduced to $(\sigma_1, \ldots)$. A mapping m is used to introduce for all variables v new variables $m(v)$ which refer to values of v in σ_0. Function $\mathcal{N}_m$ transforms all formulas in the antecedent and succedent to first order formulas restricting variables $m(v)$ and temporal formulas which are concerned with the reduced interval $(\sigma_1, \ldots)$. Fig. 4 lists the definitions of $\mathcal{N}_m$ for all basic temporal operators. Definitions for the other operators can be derived. For example

$$\mathcal{N}_m(\Box\,\varphi) = \mathcal{N}_m(\varphi) \wedge \Box\,\varphi \quad \text{and} \quad \mathcal{N}_m(\bigcirc\,\varphi) = \varphi.$$

For programs, a function $\mathcal{T}_m$ is used to decompose nonempty α into a set of $(\varphi_\alpha, \alpha')$ with formulas φ_α describing the first transition and corresponding α' which represents the remaining transition system. For example

$$\mathcal{T}_m(\underbrace{y_1 := n; y_2 := 1; \ldots}_{\alpha}) = \{(\underbrace{y_1 = m(n) \wedge y_2 = m(y_2) \wedge \ldots}_{\varphi_\alpha}, \underbrace{y_2 := 1; \ldots}_{\alpha'})\}.$$

Here, the only first transition of program α is the execution of assignment $y_1 := n$. For this transition a formula φ_α is generated. The value of y_1 in the next state is the old value $m(n)$ of n. A so called frame assumption ensures every other variable to be unchanged ($y_2 = m(y_2) \wedge \ldots$). After executing the first assignment, $y_2 := 1; \ldots$ is the remaining transition system α'. In general φ_α is the strongest post condition of the first transition of α. A complete definition of $\mathcal{T}_m$ is contained in Fig. 5. Instead

$$
\begin{array}{rcl}
\mathcal{N}_m &:& \mathrm{Fma} \mapsto \mathrm{Fma}
\end{array}
\qquad\qquad
\begin{array}{rcl}
\mathcal{L} &:& \mathrm{Fma} \mapsto \mathrm{Fma}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{N}_m(p) &:=& m(p) \\
\mathcal{N}_m(\neg\,\varphi) &:=& \neg\,\mathcal{N}_m(\varphi) \\
\mathcal{N}_m(\varphi \wedge \psi) &:=& \mathcal{N}_m(\varphi) \wedge \mathcal{N}_m(\psi) \\
\mathcal{N}_m(\mathbf{unit}) &:=& \mathbf{last} \\
\mathcal{N}_m(\varphi \frown \psi) &:=& \mathcal{N}_m(\mathcal{L}(\varphi)) \wedge \mathcal{N}_m(\psi) \\
&& \vee\ (\mathcal{N}_m(\varphi) \frown \psi) \\
\mathcal{N}_m(\varphi\ \mathbf{unless}\ \psi) &:=& \mathcal{N}_m(\psi) \\
&& \vee\ \mathcal{N}_m(\varphi) \wedge (\varphi\ \mathbf{unless}\ \psi) \\[4pt]
\mathcal{N}_m(\mathbf{E}) &:=& \mathrm{false} \\
\mathcal{N}_m(\alpha) &:=& \bigvee\{\varphi_\alpha \wedge \alpha' \mid (\varphi_\alpha, \alpha') \in \mathcal{T}_m(\alpha)\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{L}(p) &:=& p \\
\mathcal{L}(\neg\,\varphi) &:=& \neg\,\mathcal{L}(\varphi) \\
\mathcal{L}(\varphi \wedge \psi) &:=& \mathcal{L}(\varphi) \wedge \mathcal{L}(\psi) \\
\mathcal{L}(\mathbf{unit}) &:=& \mathrm{false} \\
\mathcal{L}(\varphi \frown \psi) &:=& \mathcal{L}(\varphi) \wedge \mathcal{L}(\psi) \\[4pt]
\mathcal{L}(\varphi\ \mathbf{unless}\ \psi) &:=& \mathcal{L}(\varphi) \vee \mathcal{L}(\psi) \\[4pt]
\mathcal{L}(\mathbf{E}) &:=& \mathrm{true} \\
\mathcal{L}(\alpha) &:=& \mathrm{false}
\end{array}
$$

Figure 4: Definitions of $\mathcal{N}_m$ and $\mathcal{L}$ ($\alpha \neq \mathbf{E}$)

$$
\begin{array}{rcl}
\mathcal{T}_m &:& \mathrm{Prog} \mapsto \{(\mathrm{Fma}, \mathrm{Prog})\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{T}_m(x := \tau) &:=& \{(x = \mathcal{N}_m(\tau) \wedge \mathrm{frame}(m \setminus \{x\}), \mathbf{E})\} \\
\mathcal{T}_m(\alpha; \beta) &:=& \mathcal{T}_m(\alpha); \beta \\
\mathcal{T}_m(\mathbf{if}\ \varepsilon\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta) &:=& \{(\varepsilon \wedge \mathrm{frame}(m), \alpha), (\neg\,\varepsilon \wedge \mathrm{frame}(m), \beta)\} \\
\mathcal{T}_m(\mathbf{while}\ \varepsilon\ \mathbf{do}\ \alpha) &:=& \{(\varepsilon \wedge \mathrm{frame}(m), \alpha; \mathbf{while}\ \varepsilon\ \mathbf{do}\ \alpha), (\neg\,\varepsilon \wedge \mathrm{frame}(m), \mathbf{E})\} \\
\mathcal{T}_m(\mathbf{await}\ \varepsilon\ \mathbf{do}\ x := \tau) &:=& \mathcal{N}_m(\varepsilon) \wedge \mathcal{T}_m(x := \tau) \\
\mathcal{T}_m(\alpha \parallel \beta) &:=& (\mathcal{T}_m(\alpha) \parallel \beta) \cup (\alpha \parallel \mathcal{T}_m(\beta)) \\
\mathcal{T}_m(l : \alpha) &:=& \mathcal{T}_m(\alpha)
\end{array}
$$

where

$$
\begin{array}{rcl}
\mathrm{frame}(m) &:=& \bigwedge_{v \in m, v \notin \mathbb{L}} v = m(v) \\
\mathcal{T}_m(\alpha) \parallel \beta &:=& \{(\varphi_\alpha, \alpha' \parallel \beta) \mid (\varphi_\alpha, \alpha') \in \mathcal{T}_m(\alpha)\} \\
\alpha \parallel \mathcal{T}_m(\beta) &:=& \{(\varphi_\beta, \alpha \parallel \beta') \mid (\varphi_\beta, \beta') \in \mathcal{T}_m(\beta)\} \\
\mathcal{T}_m(\alpha); \beta &:=& \{(\varphi_\alpha, \alpha'; \beta) \mid (\varphi_\alpha, \alpha') \in \mathcal{T}_m(\alpha)\} \\
\varphi \wedge \mathcal{T}_m(\alpha) &:=& \{(\varphi \wedge \varphi_\alpha, \alpha') \mid (\varphi_\alpha, \alpha') \in \mathcal{T}_m(\alpha)\}
\end{array}
$$

Figure 5: Definition of $\mathcal{T}_m$

of translating the transition system in advance, $\mathcal{T}_m$ generates formulas in predicate logic for the next possible transitions on the fly, α is symbolically executed.

For generating φ_α, function $\mathcal{T}_m$ ignores all labels. This is because the remaining transition system α' uniquely defines all label settings in the next state

$$l = \text{true} \leftrightarrow l \in lab(\alpha') \text{ for all } l \in \mathbb{L}.$$

For the second premise of rule *step* function $\mathcal{L}$ (see Fig. 4) is used to reduce the temporal formulas into first order formulas describing the last state of an interval.

As an example of symbolic execution, consider sequent

$$0 \le k \le n, y_1 = n, y_2 = 1, b = 1, s = 1, (l : \textbf{while} \ldots) \parallel (q : \textbf{while} \ldots)$$
$$\vdash \Box \text{ Prop}$$
(1)

which represents a situation in which the initial assignments of program *Binom* have been executed. Applying rule *step* on the sequent results in two premises. The first premise reads as follows

$$
\begin{aligned}
& 0 \le k' \le n', y_1' = n', y_2' = 1, b' = 1, s' = 1, \\
& \quad y_1' > n' - k' \wedge n = n' \wedge \ldots \wedge t_2 = t_2' \\
& \quad \wedge (l_1 : P(s); \ldots; l : \textbf{while} \ldots) \parallel (q : \textbf{while} \ldots) \\
& \vee \quad \neg \, y_1' > n' - k' \wedge n = n' \wedge \ldots \wedge t_2 = t_2' \\
& \quad \wedge \textbf{E} \parallel (q : \textbf{while} \ldots) \\
& \vee \quad y_2' \le k' \wedge n = n' \wedge \ldots \wedge t_2 = t_2' \\
& \quad \wedge (l : \textbf{while} \ldots) \parallel (q_1 : \textbf{await} \ldots; q : \textbf{while} \ldots) \\
& \vee \quad \neg \, y_2' \le k' \wedge n = n' \wedge \ldots \wedge t_2 = t_2' \\
& \quad \wedge (l : \textbf{while} \ldots) \parallel \textbf{E} \\
& \vdash (\text{false} \rightarrow b' = \binom{n'}{k'}) \wedge \Box \text{ Prop}.
\end{aligned}
$$

New variables $y_1', y_2', \ldots$ have been introduced to refer to the original first state. The disjunction in the antecedent corresponds to the four different transitions which are possible for the parallel statement. Either the first or the second program is executed and the condition of the executed while loop is either true or false. The premise can be automatically simplified to

$$
\begin{aligned}
& 0 \le k \le n, y_1 = n, y_2 = 1, b = 1, s = 1, \\
& \quad 0 < k \wedge (l_1 : P(s); \ldots; l : \textbf{while} \ldots) \parallel (q : \textbf{while} \ldots) \\
& \vee \, 0 = k \wedge q : \textbf{while} \ldots \\
& \vee \, 0 < k \wedge (l : \textbf{while} \ldots) \parallel (q_1 : \textbf{await} \ldots; q : \textbf{while} \ldots) \\
& \vee \, 0 = k \wedge l : \textbf{while} \ldots \\
& \vdash \Box \text{ Prop}.
\end{aligned}
$$

The second premise

$$\textbf{last}, 0 \le k \le n, y_1 = n, y_2 = 1, b = 1, s = 1, \text{false} \vdash \text{true} \rightarrow b = \binom{n}{k}$$

is trivial, because the program in the antecedent has not terminated yet. As the premises of rule *step* tend to get large, automatic simplification is essential.

Rule *step* is similar to rule *next* from [3], but considers finite intervals. Also our definitions of $\mathcal{N}_m$ are chosen such that the rule is invertible.
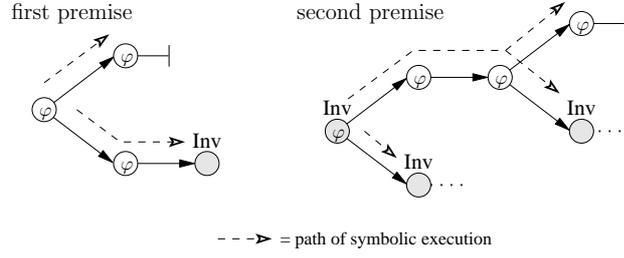
first premise   second premise

$- - -\!\!\blacktriangleright$ = path of symbolic execution

Figure 6: Visualisation of rule *always inv*

$$
\begin{array}{rcl}
\mathcal{A} & : & \mathrm{Fma} \mapsto \mathrm{Fma}
\end{array}
\qquad\qquad
\begin{array}{rcl}
\mathcal{E} & : & \mathrm{Fma} \mapsto \mathrm{Fma}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{A}(\mathrm{false}) & := & \mathrm{false} \\
\mathcal{A}(p) & := & \mathrm{true} \\
\mathcal{A}(\neg\,\varphi) & := & \neg\,\mathcal{E}(\varphi) \\
\mathcal{A}(\varphi \wedge \psi) & := & \mathcal{A}(\varphi) \wedge \mathcal{A}(\psi) \\
\mathcal{A}(\mathbf{unit}) & := & \mathbf{unit} \vee \mathbf{last} \\
\mathcal{A}(\varphi \frown \psi) & := & (\mathcal{A}(\varphi) \frown \psi) \vee \mathcal{A}(\psi) \\
\mathcal{A}(\varphi\ \mathbf{unless}\ \psi) & := & \mathcal{A}(\varphi) \wedge (\varphi\ \mathbf{unless}\ \psi) \vee \mathcal{A}(\psi) \\
\\
\mathcal{A}(\alpha) & := & \bigvee \mathcal{P}(\alpha)
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{E}(\mathrm{true}) & := & \mathrm{true} \\
\mathcal{E}(p) & := & \mathrm{false} \\
\mathcal{E}(\neg\,\varphi) & := & \neg\,\mathcal{A}(\varphi) \\
\mathcal{E}(\varphi \wedge \psi) & := & \mathcal{E}(\varphi) \wedge \mathcal{E}(\psi) \\
\mathcal{E}(\mathbf{unit}) & := & \mathbf{unit} \vee \mathbf{last} \\
\mathcal{E}(\varphi \frown \psi) & := & \mathrm{false} \\
\mathcal{E}(\varphi\ \mathbf{unless}\ \psi) & := & \mathrm{false} \\
\\
\mathcal{E}(\alpha) & := & \mathrm{false}
\end{array}
$$

Figure 7: Definitions of $\mathcal{A}$ and $\mathcal{E}$

3.2 Invariant Technique

In addition to symbolic execution, an invariant technique is needed to cope with loops
in the control flow. In the following, safety properties $\Box\,\varphi$ are considered. In calculi
for temporal logic it is common to use a global, generalized invariant which should
hold in each step. Here, a rule is proposed which allows a simpler invariant to hold
only in some states

$$
\frac{\Gamma \vdash \varphi\ \mathbf{unless}\ \mathrm{Inv}, \Delta \quad \mathrm{Inv}, \mathcal{A}(\Gamma) \vdash \varphi \wedge \bigcirc\,(\varphi\ \mathbf{unless}\ \mathrm{Inv}), \mathcal{E}(\Delta)}{\Gamma \vdash \Box\,\varphi, \Delta}\ \text{always inv.}
$$

In order to prove $\Box\,\varphi$, in a first premise φ is shown unless some state is reached in
which the invariant Inv holds. In the second premise starting from an arbitrary state
satisfying Inv, φ is shown unless the invariant again is fulfilled. Proofs advance from
one state satisfying Inv to another using symbolic execution. Figure 6 visualises a
possible scenario of rule *always inv* using computation trees. Functions $\mathcal{A}$ resp. $\mathcal{E}$ (see
Fig. 7) generalize the formulas in the antecedent resp. succedent. The definitions try
to keep as much information as possible, but information about the past is discarded.
E.g. $\Box\,\varphi$ in the antecedent is kept, but $\Diamond\,\varphi$ in the antecedent is generalized to true
and thereby discarded. For parallel programs α, a function $\mathcal{P}$ (see Fig. 8) generates
all possible execution states.

8

$$\mathcal{P} \quad : \quad \text{Fma} \mapsto \text{Fma}$$

$$
\begin{array}{rcl}
\mathcal{P}(\mathbf{E}) & := & \{\mathbf{E}\} \\
\mathcal{P}(x := \tau) & := & \{x := \tau, \mathbf{E}\} \\
\mathcal{P}(\alpha; \beta) & := & (\mathcal{P}(\alpha); \beta) \cup \mathcal{P}(\beta) \\
\mathcal{P}(\textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta) & := & \{\textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta\} \cup \mathcal{P}(\alpha) \cup \mathcal{P}(\beta) \\
\mathcal{P}(\textbf{while } \varepsilon \textbf{ do } \alpha) & := & (\mathcal{P}(\alpha); \textbf{while } \varepsilon \textbf{ do } \alpha) \cup \{\mathbf{E}\} \\
\mathcal{P}(\textbf{await } \varepsilon \textbf{ do } \alpha) & := & \{\textbf{await } \varepsilon \textbf{ do } \alpha, \mathbf{E}\} \\
\mathcal{P}(\alpha \parallel \beta) & := & \mathcal{P}(\alpha) \parallel \mathcal{P}(\beta) \\
\mathcal{P}(l : \alpha) & := & (\mathcal{P}(\alpha) \setminus \{\alpha\}) \cup \{l : \alpha\}
\end{array}
$$

Figure 8: Definition of $\mathcal{P}$

$$
\begin{aligned}
\text{Inv} \equiv \quad & l \wedge q_2 \vee l_2 \wedge q \vee l_2 \wedge q_6 \vee l_5 \wedge q_2 \qquad \textit{execution states which satisfy invariant} \\
\wedge \quad & 0 \le k \le n \\
\wedge \; & n - k \le y_1 \le n \\
\wedge \; & 1 \le y_2 \le k \\
\wedge \; & \neg\, q \rightarrow y_1 + y_2 \le n \\
\wedge \; & r = l_2 \; ? \; 1; \, 0 \\
\wedge \; & b = \frac{\Pi(y_1 + (l_5 \; ? \; 0 \; ; \; 1), n)}{\Pi(1, y_2 - (q_6 \; ? \; 0 \; ; \; 1))}
\end{aligned}
$$

$\left.\begin{array}{r}\\[6em]\end{array}\right\}$ *properties which should be invariant*

Figure 9: Local invariant for $Binom$[3]

Our example of Fig. 1 contains loops. The basic idea for verifying sequent (1) is to give an invariant which needs to hold only at some selected execution states. The invariant of Fig. 9 is satisfied only if the parallel programs are at labels (l, q_2), (l_2, q), (l_2, q_6), or (l_5, q_2). With this invariant, rule *always inv* results in the following simplified second premise

$$
\begin{aligned}
\text{Inv}, & \\
& (l : \textbf{while } \ldots) \parallel (q_2 : P(r); \ldots) \\
\vee \; & (l_2 : t_1 := b * y_1; \ldots) \parallel (q : \textbf{while } \ldots) \\
\vee \; & (l_2 : t_1 := b * y_1; \ldots) \parallel (q_6 : y_2 := y_2 + 1; \ldots) \\
\vee \; & (l_5 : y_1 := y_1 + 1; \ldots) \parallel (q_2 : P(r); \ldots) \\
\vdash \; & \text{Prop} \wedge \bigcirc (\text{Prop } \textbf{unless } \text{Inv}).
\end{aligned}
$$

Starting from four selected execution states, symbolic execution is used to execute the programs unless an execution state is reached in which the invariant again is satisfied. The four execution states for the invariant are chosen such that no loops remain in the control flow while going from one state satisfying Inv to another.

Rule *always inv* can be seen as a generalization of the invariant rule of Hoare logic [13]. Hoare requires an invariant to always hold at the beginning of the execution of the body, for *always inv* the invariant should hold at arbitrarily selectable execution states. The proposed rule can only be applied to safety properties $\Box \, \varphi$. A variant for liveness properties $\Diamond \, \varphi$ can be obtained in analogy to a Hoare rule for total

---

[3]$\varphi \; ? \; \tau_1; \tau_2$ is an abbreviation for τ_1 resp. τ_2, if φ is true resp. false.

$$\text{Inv}_g \equiv \exists\, y_1^*, y_2^*, b^* \,.$$
$$0 \le k \le n$$
$$\wedge\ y_1^* = (l_4 \vee l_5)\ ?\ \ y_1 - 1\ ;\ y_1$$
$$\wedge\ y_2^* = (q_5 \vee q_6)\ ?\ \ y_2 + 1\ ;\ y_2$$
$$\wedge\ b = \frac{\Pi(y_1^*+1,n)}{\Pi(1,y_2^*-1)}$$
$$\wedge\ l_3 \rightarrow t_1 = b \cdot y_1^*$$
$$\wedge\ q_4 \rightarrow t_2 = b / y_2^*$$
$$\wedge\ q_2 \vee q_3 \vee q_4 \rightarrow y_1^* + y_2^* \le n$$
$$\wedge\ l_6 \rightarrow y_1 = n - k$$
$$\wedge\ y_1 \ge n - k + ((l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5)\ ?\ 1\ ;\ 0)$$
$$\wedge\ q_7 \rightarrow y_2 = k + 1$$
$$\wedge\ y_1 \le k + 1 - ((q_1 \vee q_2 \vee q_3 \vee q_4 \vee q_5 \vee q_6)\ ?\ 1\ ;\ 0)$$
$$\wedge\ l_6 \wedge q_7 \rightarrow b = \frac{\Pi(n-k+1,n)}{\Pi(1,k)}$$
$$\wedge\ \neg\, ((l_2 \vee l_3 \vee l_4) \wedge (q_3 \vee q_4 \vee q_5))$$

Figure 10: global invariant

correctness. A similar invariant technique for arbitrary temporal formulas is still open.

4 Comparison

In this section, we compare our invariant technique to the global invariant technique used in [15], the same book we have taken the binomial coefficient example from. In order to show a property $\square\,\varphi$ for a given global transition system, in [15] a technique based on a global invariant is used. φ is generalized into an invariant Inv, which has to i) imply φ, ii) hold in the initial state, and iii) be invariant to every single transition. In our notation this rule could be written as follows

$$\frac{\text{Inv} \vdash \varphi \quad \Gamma \vdash \text{Inv}, \Delta \quad \mathcal{A}(\Gamma), \text{Inv} \vdash \bigcirc \varphi, \mathcal{E}(\Delta)}{\Gamma \vdash \square\,\varphi, \Delta}\ \text{global inv}$$

and the invariant from [15] for example *Binom* translates to formula Inv_g contained in Fig. 10.

Our invariant technique is based on local invariants. As the global invariant has to hold in all states, it contains information that is unnecessary in a local invariant. Thus, a global invariant will be longer and much more detailed (and therefore harder to find) than a local invariant. For example, information about the values of the auxiliary variables t_1 and t_2 can be omitted in the local invariant. Of course there are techniques for determining invariants automatically (see e.g. [5]) and these techniques can be adopted to our setting, as a local invariant is implied by a global one. It is still to be examined, though, how large the advantage of using smaller, local invariants will be, if a technique for automatically computing invariants is applied.

In our setting, not all statements in program *Binom* have to be labelled. Labels are needed only for those program locations that are referenced in the local invariant.

Furthermore, symbolic execution yields proofs that are easier to understand as they follow the control flow of the executed program.

However, proofs will become very large when symbolic execution is used without further means. The proof for the binomial coefficient example takes several thousand steps in KIV, when pure symbolic execution is used. Although the proofs can be automated to a very high degree (in the above example the only interaction required is the application of the invariant rule), it is necessary to reduce the complexity of the proof.

When two programs α and β are executed in parallel, there are a quadratic number of transitions, but an exponential number of paths through the programs, since the first step to be executed can be one from either α or β. Often it does not matter, if we first execute a step from α, followed by a single step from β, or vice versa – the resulting state is equivalent. Yet, as we got there on two different paths, two separate branches in the proof result, which doubles the size of the proof for that state.

Therefore, proofs done with pure symbolic execution are highly redundant. Removing this redundancy yields smaller proofs. Possible ways to do this are the use of lemmas, selecting more execution states for the local invariant to hold at, and, foremost, sequencing. The use of additionally selected execution states leads to shorter program fragments to be executed between two states satisfying the invariant and blowup of proof size is reduced. On the other hand, the invariant is more complex. If all possible execution states are selected, we obtain rule *global inv* as a special case.

A more complex invariant results in a more difficult interaction. As we want the degree of automation to be as high as possible, we need a means to automatically eliminate the redundancy from the proof. For this we favour sequencing. The schematic sequencing rule is as follows

$$\frac{\Gamma,\ \alpha_1;(\alpha_2 \parallel \beta)\ \vdash\ \Delta\quad\ldots}{\Gamma,\ (\alpha_1;\alpha_2)\parallel \beta\ \vdash\ \Delta}$$

By choosing *one* of the parallel programs whose first step will be executed next, sequencing allows us to restrict the (explicit) proof to a subset of all possible computations. To ensure the correctness of the sequencing step, i.e. to establish the property for those computations that start with a different first step, additional premises are generated (denoted in the above figure by '...'). These premises vary with the formula to be proven.

Sequencing is the technique we will use most, since it both yields the best results and its use can be automated. In the *Binom* example, we were able to use sequencing for most of the execution states with no communication between the two concurrent programs. Thus, we could avoid the exponential blowup and the size of the proof for the example could be reduced to just 800 proof steps – with only one single interaction.

5   Conclusion/Future Work

We have introduced an interactive proof method for verifying temporal properties of large concurrent systems with infinite states which cannot be proven with automatic techniques. Our method is based on symbolic execution which gives very intuitive proofs as they follow the control flow of the considered transition system. An implementation in KIV has shown that symbolic execution can be automated to a large

extent. In comparison with other techniques much simpler invariants are sufficient which need to hold only at a small number of execution states. However, proofs may become very large as the number of execution paths for parallel programs is exponential. In order to avoid explosion of proof size we propose to use sequencing as often as possible as this concept can be automated best. Sequencing is subject to further examination, its relation to the concept of blocking, which is used in [1] [15], is also of interest.

The paper only discussed inductive proofs of safety properties. Arbitrary temporal formulas may also state liveness properties. In order to verify liveness, fairness assumptions for the interleaving of programs are required. In principle our technique is also applicable to the verification of liveness. For this, the induction technique has to be modified and an appropriate way of expressing fairness needs to be adopted. This is work in progress. We have evidence that symbolic execution with induction can be used to verify arbitrary temporal formulas of different temporal logics for parallel programs and is not restricted to a subset as in [17].

Furthermore the proof method is not restricted to parallel programs. For example state charts [11] [12] could also be used to describe transition systems. As prerequisite an operational semantics for the formalism is needed (similar to relation $\longrightarrow$ of section 2). The operational semantics is the basis for defining $\mathcal{T}_m$ (see Fig. 5) which should provide all possible next transitions together with the remaining transition system.

In this paper, interval temporal logic has been used to formulate properties. It should also be possible to apply the proof method to duration calculus (DC) [8] which can be used to express real time properties. Although DC is based on a continuous semantics and does not define a next step operator $\bigcirc \varphi$, the requirement of finite variability guarantees that the system advances in discrete steps and symbolic execution can be applied.

References

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

[2] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS. Springer, 2000.

[3] S. Biundo, D. Dengler, and J. Koehler. Deductive planning and plan reuse in a command language environment. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 628–632, 1992.

[4] N. Bjorner, A. Brown, and M. A. Collon et al. Verifying temporal properties of reactive systems: A step tutorial. In *Formal Methods in System Design*, 2000.

[5] N. Bjorner, A. Brown, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings, Fifth Annual*

IEEE Symposium on Logic in Computer Science, pages 428–439, Philadelphia, Pennsylvania, 4–7 June 1990. IEEE Computer Society Press.

[7] R. M. Burstall. Program proving as hand simulation with a little induction. *Information processing 74*, pages 309–312, 1974.

[8] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, December 1991.

[9] CoFI: The Common Framework Initiative. Casl — the CoFI algebraic specification language tentative design: Language summary, 1997. http://www.brics.dk/Projects/CoFI.

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

[11] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3), 1987.

[12] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[13] C.A.R. Hoare. An axiomatic basis for computer programming. *COMM ACM*, pages 576–580, 1969.

[14] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), 1994.

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

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

[17] A. U. Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, 1993.