Verification by Augmented Abstraction: The Automata-Theoretic View

Yonit Kesten* Amir Pnueli† Moshe Y. Vardi ‡

November 23, 2000

*Contact author: Dept. of Communication Systems Engineering, Ben Gurion University, Israel,

†Dept. of Applied Mathematics and Computer Science, the Weizmann Institute of Science, Israel. This research was supported in part by the Minerva Center for Verification of Reactive Systems, by a grant from the U.S.-Israel Binational Science Foundation, by an NSF grant CCR-970061 and by a grant from the Intel Corporation

‡Dept. of Computer Science, Rice University, Houston, TX 7725-1892, U.S.A. Work Partly done when this author was a Varon Visiting Professor at the Weizmann Institute of Science and partly supported by NSF grants CCR-9700061 and CCR-9988322 and by a grant from the Intel Corporation.

Verification by Augmented Abstraction

Contact Author:
Yonit Kesten
Department of Communication Systems Engineering
Ben-Gurion University
Beer-Sheva
Israel
e-mail: ykesten@bgumail.bgu.ac.il
Tel: 972-7-6469-988

Abstract.

The paper deals with the proof method of *verification by finitary abstraction* (VFA), which presents an alternative approach to the verification of (potentially infinite-state) reactive systems. We assume that the negation of the property to be verified is given by the user in the form of an infinite-state non-deterministic *Büchi disctrete system* (BDS). The method consists of a two-step process by which, in a first step, the system and its (negated) specification are combined into a single infinite-state *fair discrete system* (FDS, which is similar to a BDS but with Streett acceptance conditions), which is abstracted into a finite-state automaton. The second step uses model checking to establish that the abstracted automaton is infeasible, i.e., has no computations.

The VFA method can be considered as a viable alternative to verification by temporal deduction, which, up to now, has been the main method generally applicable for verification of infinite-state systems.

The paper presents a general recipe for an FDS abstraction, which is shown to be *sound*, where soundness means that infeasibility of the abstracted FDS implies infeasibility of the unabstracted one, implying in turn the validity of the property over the concrete (infinite-state) system. To make the method applicable for the verification of liveness properties, pure abstraction is sometimes no longer adequate. We show that by augmenting the system with an appropriate (and standardly constructible) *progress monitor*, we obtain an augmented system, whose computations are essentially the same as those of the original system, and which may now be abstracted while preserving the desired liveness properties. We refer to the extended method as *verification by augmented abstraction* (VAA).

We then proceed to show that the VAA method is sound and complete for proving all properties whose negations are expressible by a BDS. Given that every LTL property can be translated to a BDS, this establishes that the VAA method is sound and complete for proving the validity of all LTL properties, including both safety and liveness.

1 Introduction

When verifying temporal properties of reactive systems, the common wisdom is: if it is finite-state, model check it, otherwise use temporal deduction, supported by theorem provers such as STeP, PVS, etc. The study of abstraction as an aid to verification demonstrated that, in some interesting cases, one can abstract an infinite-state system into a finite-state one. This suggests an alternative approach to the temporal verification of infinite-state systems: abstract first and model check later. This general idea can be developed and applied in various frameworks that may differ from one another by the formalisms used for computing the abstraction and model checking the resulting abstraction. In all of these approaches, we consider a system $\mathcal{D}$ presented as a *fair discrete system* (FDS) and a specification given by a *linear temporal logic* (LTL) formula ψ.

The work reported in [KP99b], presented a version of the *verification by finitary abstraction* (VFA) method in which the separation between the system and its specification was maintined throughout the abstraction process. The method there was based on a joint abstraction of the reactive system $\mathcal{D}$ and its specification ψ. The unique features of the abstraction method of [KP99b] are that it takes full account of all the fairness assumptions (including strong fairness) associated with the system $\mathcal{D}$ and can, therefore, establish liveness properties, in contrast to most previous abstraction approaches that can only support verification of safety properties.

Since the presentation in [KP99b] worked directly with the temporal formula ψ, it was necessary to present two different recipes for abstraction: the first dealing with the *fair discrete system* $\mathcal{D}$ and the other showing how to abstract a temporal formula. As a result, the presentation was more involved and the proof of completeness of the approach became specially complex, due to the need to deal separately with the two formalisms which, in principle, are very close to one another. This additional complexity may obstruct the inherent simplicity of the ideas underlying the VFA method.

In this paper, we chose to develop the VFA method in a more homogenous and uniform framework, in which both the verified system $\mathcal{D}$ and its specification are presented as FDS's, which are, in principle, ω-automata (Streett automata to be precise) extended syntactically to deal with infinite-state systems. Indeed, couched in an automata-theoretic framework, the presentation is very much simplified and the basic ideas become clearer.

We start with $\mathcal{D}$, the FDS representing the system to be verified, and a *Büchi discrete system* (BDS) $T_{\neg\psi}$, which is an FDS with Büchi acceptance conditions, representing the complemented property $\neg\psi$, namely all the sequences violating the property, or all counter examples. For users prefering linear temporal logic as specification language, we give references in Section 3 to the construction of $T_{\neg\psi}$ for a given formula ψ.

As usual in the automata-theoretic approach, we reduce the problem of verifying $\mathcal{D} \models \psi$ to proving that the BDS $\mathcal{B} :\ \mathcal{D} \,|||\, T_{\neg\psi}$, formed by taking the *synchronous parallel composition* of $\mathcal{D}$ with $T_{\neg\psi}$ is *infeasible*, i.e., has no computations.

We first provide a sound recipe for the application of the method of *verification by finitary abstraction* (VFA). That is, given an arbitrary state mapping α which maps concrete to abstract states, we show how to define the abstracted version $\mathcal{B}^{\alpha}$, such that if $\mathcal{B}^{\alpha}$ is infeasible then so is $\mathcal{B}$, establishing that $\mathcal{D} \models \psi$. In the case that α maps all concrete variables into abstract variables ranging over finite domains, $\mathcal{B}^\alpha$ will be a finite-state system, and the infeasibility of $\mathcal{B}^\alpha$ can be verified by model checking. Some interesting examples of abstractions of an infinite-state system into a finite-state one have been presented in [BBM95] and [KP98].

Applying the method of finitary abstraction to the verification of liveness properties, we find that, sometimes, pure abstraction is no longer adequate. For these cases, it is possible to construct an additional module M, to which we refer as a *progress monitor*, such that the augmented system $\mathcal{D} \parallel\!\mid M$ has essentially the same set of computations as the original $\mathcal{D}$ and can be abstracted in a way that preserves the desired liveness property. We refer to this extended proof method as the method of *verification by augmented abstraction* (VAA).

In Section 7 we formulate the VAA method in the automata-theoretic framework and show that the method is sound. That is, for every abstraction mapping α, if the abstracted composed system $(\mathcal{D} \parallel\!\mid M \parallel\!\mid T_{\neg\psi})^\alpha$ is infeasible, and the monitor M does not constrain the computations of $\mathcal{D}$ (effective sufficient conditions for this are provided), then we can safely infer the infeasibility of the original system $\mathcal{D} \parallel\!\mid T_{\neg\psi}$.

Section 8 is dedicated to the proof of *completeness* of the VAA method in the automata-theoretic framework. In this section, we show that if $\mathcal{D} \parallel\!\mid T_{\neg\psi}$ is infeasible, then there exists a monitor M that does not constrain the computations of $\mathcal{D}$ and a finitary abstraction mapping α, such that $(\mathcal{D} \parallel\!\mid M \parallel\!\mid T_{\neg\psi})^\alpha$ is infeasible.

As will be shown in the next subsection, the idea of using abstraction for simplifying the task of verification is certainly not new. Even the observation that, in many interesting cases, infinite-state systems can be abstracted into finite-state systems which can be model checked has been made before. In [KP99b], we show that for some verification tasks involving liveness, pure abstraction is inadequate, and devise the method of *verification by augmented abstraction* (VAA). We then establish completeness of the VAA method.

The main contributions of the current paper can be summarized as follows:

- Reformulation of the method of verification by augmented abstraction within the automata-theoretic framework.
- Establishing completeness of the VAA method within this framework.

1.1 Related Work

Most previous works on verification by finitary abstraction follow the work on verification in which the system is specified by transition systems and the verified property is specified by one of the temporal logics like LTL, CTL, μ-calculus, etc. In these works, the system and the property are abstracted separately, using different methodologies for abstracting the system and the properties specified in these logics. There has been an extensive study of the use of data abstraction techniques in these frameworks, mostly based on the notions of *abstract interpretation* ([CC77, CH78]). See for example [CGL94, CGL96, DGG97, LGS$^+$95, BBM95]. All of these methods are only applied for the verification of safety properties. Liveness, and therefore fairness, are not considered.

A deductive methodology for proving temporal properties over infinite state systems is presented in [MP91a]. The methodology is based on a set of proof rules, each devised for a class of temporal formulas. This methodology is proved to be complete, relative to the underlying assertion language.

Verification diagrams (VD), presented in [MP94], provide a finite graphical representation of the deductive proof rules, which can be viewed as a finite abstraction of the verified system, with respect to the verified property.

In [BMS95, MBSU98], the notion of a verification diagram is generalized (GVD), allowing a uniform verification of arbitrary temporal formulas. The GVD method is also shown to be sound and complete. The abstraction constructed by this method can be viewed as an ω-automaton with either Streett ([BMS95]) or Muller ([MBSU98]) acceptance condition.

A dual method to VD is the deductive model checking (DMC) presented in [SUM96]. Similar to VD, this method tries to verify a temporal property over an infinite state system, using a finite graph representation. The procedure starts with the temporal tableau for $\neg\varphi$, which is repeatedly refined until either a counter example is found or the property is proved. The method is shown to be complete in [SUM99].

Moving to the automata-theoretic framework, the problem of verification is reduced to the problem of emptiness of (possibly infinite-state) automata. Verification by finitary abstraction in the automata-theoretic framework means abstracting a possibly infinite state automaton into a finite state automaton, preserving nonemptiness. Abstraction in the automata framework has been studied as a state-space minimization technique [Kur95], but the focus there is on soundness and not on completeness.

A conference version of this paper appeared in [KP99a].

2 A Computational Model: Fair Discrete Systems

We assume an underlying assertion language $\mathcal{L}$ that contains the predicate calculus augmented with fixpoint operators.[1] We assume that $\mathcal{L}$ contains interpreted symbols for expressing the standard operations and relations over the integers.

Let p be an assertion and V be the set of free variables in p. Let Σ denote the set of interpretations over V. We say that *p holds* on $s \in \Sigma$, denoted $s \models p$, if $p[s] = \mathrm{T}$. An assertion p is called *satisfiable* if it holds on some $s \in \Sigma$. An assertion p is called *valid*, denoted $\models p$, if it holds on all $s \in \Sigma$. Two assertions p and q are defined to be *equivalent*, denoted $p \sim q$, if $p \leftrightarrow q$ is valid, i.e $s \models p$ iff $s \models q$, for all $s \in \Sigma$.

As a computational model for reactive systems, we take the model of a *fair discrete system* (FDS), which is a slight variation on the model of *fair transition system* (FTS) [MP95]. The FDS model was first introduced in [KPR98] under the name "Fair Kripke Structure". The main difference between the FDS and FTS models is in the representation of fairness constraints.

An FDS $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ consists of the following components.

[1] As is well known ([LPS81],) a first-order language is not adequate to express the assertions necessary for (relative) completeness of a proof system for proving validity of temporal properties of reactive programs (which in this paper are specified by automata). The use of minimal and maximal fixpoints for relative completeness of the proof rules for liveness properties is discussed in [MP91a], based on [SdRG89]. However, the fixpoints are not needed for the assertion language used to specify the components of an FDS (Θ, ρ, $\mathcal{J}$ and $\mathcal{C}$) or the set of its reachable states (see section 4).

- $V = \{u_1, ..., u_n\}$: A finite set of typed *system variables*, containing data and control variables. The set of *states* (interpretation) over V is denoted by Σ. We denote by $s[u]$ the value assigned to $u \in V$ by state s.
- Θ : The *initial condition* – an *assertion* characterizing the initial states.
- ρ : A *transition relation* – an assertion $\rho(V, V')$, relating the values V of the variables in state $s \in \Sigma$ to the values V' in a $\mathcal{D}$-successor state $s' \in \Sigma$.
- $\mathcal{J} = \{J_1, \ldots, J_k\}$: A set of *justice* requirements (*weak fairness*). The justice requirement $J \in \mathcal{J}$ is an assertion, intended to guarantee that every computation contains infinitely many J-states (states satisfying J).
- $\mathcal{C} = \{\langle p_1, q_1\rangle, \ldots \langle p_n, q_n\rangle\}$: A set of *compassion* requirements (*strong fairness*). The compassion requirement $\langle p, q\rangle \in \mathcal{C}$ is a pair of assertions, intended to guarantee that every computation containing infinitely many p-states also contains infinitely many q-states.

A state s' is said to be a $\mathcal{D}$-*successor* of a state s if $\langle s, s'\rangle \models \rho(V, V')$, where we interpret V over s and V' as the state variables of s'. For an assertion $p = p(V)$, we denote by p' the assertion $p(V')$.
A *computation* of an FDS $\mathcal{D}$ is an infinite sequence of states $\sigma : s_0, s_1, s_2, ...$, satisfying the requirements:

- *Initiality:* s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution:* For each $j = 0, 1, ...$, the state s_{j+1} is a $\mathcal{D}$-successor of s_j.
- *Justice:* For each $J \in \mathcal{J}$, σ contains infinitely many J-positions
- *Compassion:* For each $\langle p, q\rangle \in \mathcal{C}$, if σ contains infinitely many p-positions, it must also contain infinitely many q-positions.

We denote by $Comp(\mathcal{D})$ the set of all computations of $\mathcal{D}$. An FDS $\mathcal{D}$ is called *feasible* if $Comp(\mathcal{D}) \neq \emptyset$. The feasibility of a finite-state FDS can be checked algorithmically, as presented in [LP84, Eme85], and adapted to symbolic model checking in [CGH97, KPR98]. A state is called $\mathcal{D}$-*reachable* if it appears in some computation of $\mathcal{D}$.

Let $U \subseteq V$ be a set of variables. Let σ be an infinite sequence of states. We denote by $\sigma \Downarrow_U$ the *projection* of σ onto the subset U. We denote by $Comp(\mathcal{D}) \Downarrow_U$ the set of computations of $\mathcal{D}$, projected onto the set of variables U. Let $\mathcal{D}_1\colon \langle V_1, \Theta_1, \rho_1, \mathcal{J}_1, \mathcal{C}_1\rangle$ and $\mathcal{D}_2\colon \langle V_2, \Theta_2, \rho_2, \mathcal{J}_2, \mathcal{C}_2\rangle$ be two FDS's and $U \subseteq V_1 \cap V_2$. We say that D_1 is U-*equivalent* to $\mathcal{D}_2$ ($\mathcal{D}_1 \sim_U \mathcal{D}_2$) if $Comp(\mathcal{D}_1)\Downarrow_U = Comp(\mathcal{D}_2)\Downarrow_U$.

All our concrete examples are given in SPL (Simple Programming Language), which is used to represent concurrent programs (e.g., [MP95], [MAB+94b]). Every SPL program can be compiled into an FDS in a straightforward manner (see [KPR98]). Assuming that location ℓ_j appears within the program for process P_i, the predicates at_ℓ_j stands for the assertions $\pi_i = j$, where π_i is the control variable denoting the current location within P_i.

2.1 Synchronous Parallel Composition

Let $\mathcal{D}_1 : \langle V_1, \Theta_1, \rho_1, \mathcal{J}_1, \mathcal{C}_1 \rangle$ and $\mathcal{D}_2 : \langle V_2, \Theta_2, \rho_2, \mathcal{J}_2, \mathcal{C}_2 \rangle$ be two fair discrete systems. We define the *synchronous parallel composition* of $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathcal{D}_1 \parallel\!| \mathcal{D}_2$, to be the system $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where,

$$\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl} V & = & V_1 \cup V_2 & \Theta & = & \Theta_1 \wedge \Theta_2 & \rho & = & \rho_1 \wedge \rho_2 \\ \mathcal{J} & = & \mathcal{J}_1 \cup \mathcal{J}_2 & \mathcal{C} & = & \mathcal{C}_1 \cup \mathcal{C}_2 & & & \end{array}$$

As implied by the definition, each of the basic actions of system $\mathcal{D}$ consists of the joint execution of an action of $\mathcal{D}_1$ and an action of $\mathcal{D}_2$. We can view the execution of $\mathcal{D}$ as the *joint execution* of $\mathcal{D}_1$ and $\mathcal{D}_2$. The main, well established, use of synchronous parallel composition is for coupling a system $\mathcal{D}$ with an FDS representing a (negated) property over $\mathcal{D}$, and then checking the feasibility of the combined system, as will be shown in the following sections. In this work, synchronous composition is also used for coupling the system with a *monitor*, used to ensure completeness of the data abstraction methodology. We remind the reader that the concurrent composition of several SPL processes is an *asynchronous* composition based on interleaving, which is not presented here.

2.2 From FDS to JDS

An FDS with no compassion requirements is called a *just discrete system* (JDS).

Let $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ be an FDS such that $\mathcal{C} = \{(p_1, q_1), \ldots, (p_m, q_m)\}$ and $m > 0$. We define a JDS $\widetilde{\mathcal{D}}, : \langle \widetilde{V}, \widetilde{\Theta}, \widetilde{\rho}, \widetilde{\mathcal{J}}, \widetilde{\mathcal{C}} : \emptyset \rangle$ which is V-equivalent to $\mathcal{D}$ as follows. First we construct m similar JDS's, $\mathcal{D}_1, \ldots, \mathcal{D}_m$, one for each compassion requirement $(p_i, q_i) \in \mathcal{C}$. The JDS $\mathcal{D}_i$ representing a compassion requirement (p_i, q_i), is presented in Fig. 1.

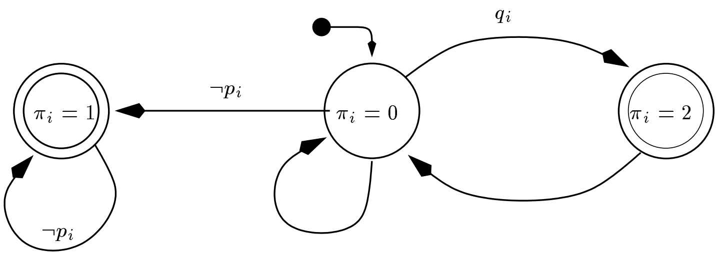

Figure 1: A JDS $\mathcal{D}_i$ for a single compassion requirement $(p_i, q_i) \in \mathcal{C}$

Each $\mathcal{D}_i$ consists of the components $V_i = V \cup \{\pi_i : [0..2]\}$, initial condition $\Theta_i : (\pi_i = 0)$, a single justice requirement $J_i : (\pi_i > 0)$ and no compassion requirements. The JDS $\widetilde{\mathcal{D}}$ is given by $\widetilde{\mathcal{D}} : \quad \mathcal{D} \parallel\!| \mathcal{D}_1 \parallel\!| \ldots \parallel\!| \mathcal{D}_m$.

The transformation of an FDS to a JDS follows the transformation of Streett automata to generalized Büchi automata (see [Cho74] for finite state automata, [Var91] for infinite state automata). More efficient representation of the resulting JDS can be obtained. For example, we can reduce the size of the augmenting component from 3^m as implied by the construction of Fig. 1 into $m \cdot 2^m$. Unfortunately, it will always be exponential in m.

2.3 From JDS to BDS

A JDS with a single justice requirement is called a *Büchi discrete system* (BDS). Let $\mathcal{D}:\langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} : \emptyset\rangle$ be a JDS such that $\mathcal{J} = \{J_1, \ldots, J_k\}$ and $k > 1$. We define a BDS $\mathcal{B}:\langle V_{\mathcal{B}}, \Theta_{\mathcal{B}}, \rho_{\mathcal{B}}, \mathcal{J}_{\mathcal{B}} : \{J\}, \mathcal{C}_{\mathcal{B}} : \emptyset\rangle$ that is V-equivalent to $\mathcal{D}$, as follows:

- $V_{\mathcal{B}} = V \cup \{u\}$, where u is a new variable not in V, interpreted over $[0..k]$.
- $\Theta_{\mathcal{B}} : \quad \Theta \ \wedge \ u = 0$.
- $$\rho_{\mathcal{B}} \ : \ \rho(V, V') \ \wedge \ \bigvee_{i=0}^{k} (u = i) \ \wedge \ u' = \begin{bmatrix} \textbf{case} & & \\ \ \ i = 0 : 1 & & ; \\ \ \ J_i \quad : (u+1) \ mod \ (k+1) & & ; \\ \ \ true \ : u & & ; \\ \textbf{esac} & & \end{bmatrix}$$
- $\mathcal{J}_{\mathcal{B}} = \{J\}$, where J is the single justice requirement $J:(u = 0)$.

The transformation of a JDS to a BDS follows the transformation of generalized Büchi automata to Büchi automata [Cho74].

3 Requirement Specification Languages

One of the well established languages for specifying properties of reactive systems is *linear temporal logic* (LTL) [MP91b, MP95]. The validity of an LTL property over an *infinite state* system is verified by deductive methods. A (relatively) complete deductive proof method for LTL properties is presented in [MP91a], and has been implemented into a verification tool in [MAB+94a]. The validity of a propositional LTL property over a *finite state* system can be verified algorithmically [LP85, VW86]. A symbolic, BDD-based, algorithmic method is discussed in [BCM+92, CGH97, KPR98].

Given a propositional LTL formula ψ, one can construct a finite ω-automaton that accepts precisely the computations satisfied by ψ [VW94]. Most algorithmic methods for the verification of LTL properties over finite state systems, are based on this observation, transforming (the negation of) the LTL property into an w-automaton [VW86, LP85].

In this work we take the automata-theoretic approach, using Büchi automata, extended to variables that can range over infinite domains, as the specification language. Automata were first introduced as a specification language for concurrent systems by Wolper in [Wol83], where finite-state ω-automata are used. The use of infinite-state ω-automata for the specification of infinite-state systems, is discussed in [Var91].

Let $\mathcal{D}$ be an FDS and $\mathcal{B}$ be a BDS representing a property over $\mathcal{D}$. We say that the system $\mathcal{D}$ satisfies the property $\mathcal{B}$ iff $Comp(\mathcal{D}) \subseteq Comp(\mathcal{B})$, which is equivalent to $Comp(\mathcal{D}) \cap Comp(\bar{\mathcal{B}}) = \emptyset$, where $\bar{\mathcal{B}}$ is the complement of $\mathcal{B}$. Since, unlike finite-state w-automata, infinite-state w-automata are not closed under complementation (see [Sis89]), we assume that the negation of the property to be verified is given by the user in the form of a BDS. Users prefering LTL as their specification language, can use the systematic transformation of a general LTL formula into a BDS as described in [KP99b]. Given an LTL formula ψ, [KP99b] construct a *tester* $T_{\neg\psi}$ for ψ, which is a BDS characterizing all the sequences which violate ψ.

Some of the tools that have been developed for automatic verification of (finite state) reactive systems, use this approach, representing the negated property directly as a Büchi automaton ([Hol97, Kur95]).

4 Verifying Büchi Discrete Systems

4.1 Verification Reduced to Infeasibility

Let $\mathcal{D}$ be an FDS and $T_{\neg\psi}$ be a BDS representing the negated property. The verification problem $Comp(\mathcal{D}) \cap Comp(T_{\neg\psi}) \stackrel{?}{=} \emptyset$ is reduced to an infeasibility problem of a BDS, as follows:

- Construct the synchronous parallel composition $\mathcal{D} \||| T_{\neg\psi}$.
- Transform the FDS $\mathcal{D} \||| T_{\neg\psi}$ into an equivalent BDS $\mathcal{B}_{(\mathcal{D},\neg\psi)}$.

Claim 1 $\quad Comp(\mathcal{D}) \cap Comp(T_{\neg\psi}) = \emptyset \quad$ *iff* $\quad Comp(\mathcal{B}_{(\mathcal{D},\neg\psi)}) = \emptyset$
i.e., iff $\mathcal{B}_{(\mathcal{D},\neg\psi)}$ is infeasible.

(see [Var91] for the proof).

4.2 Infeasibility of Büchi Discrete Systems

In the following, we present a general proof method for establishing that a BDS is infeasible.

A *well-founded domain* $(\mathcal{W}, \prec)$ consists of a set $\mathcal{W}$ and a total ordering relation $\prec$ over $\mathcal{W}$ such that there does not exist an infinitely descending sequence, i.e., a sequence of the form

$$a_0 \succ a_1 \succ a_2 \succ \cdots ,$$

A *ranking function* for an FDS $\mathcal{D}$ is a function δ mapping the states of $\mathcal{D}$ into a well-founded domain.

The standard approach to prove infeasibility of a BDS $\mathcal{B}: \langle V, \Theta, \rho, \mathcal{J} : \{J\}, \mathcal{C} : \emptyset\rangle$, is to define a ranking function δ for $\mathcal{B}$. The ranking function is required to satisfy the conditions that every transition of $\mathcal{B}$ does not increase the rank and every transition into a state satisfying J, the single justice requirement of $\mathcal{B}$, decreases the rank. The (possibly infinite) set of reachable states of $\mathcal{B}$ can be characterized (or over-approximated) by an inductive assertion φ. The infeasibility of $\mathcal{B}$ can then be derived from rule WELL, presented in Fig. 2. Rule WELL is sound and complete, relative to the assertional validity, for proving emptiness of a BDS. Soundness of the rule means that, given a BDS $\mathcal{B}$, if we can find a ranking function δ and an assertion φ, such that φ and δ satisfy the three premises W1–W3, then $\mathcal{B}$ is indeed infeasible. To see this, assume, to the contrary, that $\mathcal{B}$ is feasible. Then $\mathcal{B}$ has an infinite computation $\sigma: s_0, s_1, \ldots$, such that $s_i \models J$ for infinitely many states s_i in σ. Then, from

For an assertion φ,
a single justice requirement J,
a well founded domain $(\mathcal{W}, \prec)$,
and a ranking function $\delta : \Sigma_V \mapsto \mathcal{W}$

$$\begin{array}{lll}
\text{W1.} & \Theta & \rightarrow \quad \varphi \\[6pt]
\text{W2.} & \rho \wedge \varphi & \rightarrow \quad \varphi' \wedge \delta' \preceq \delta \\[6pt]
\text{W3.} & \rho \wedge \varphi \wedge J' & \rightarrow \quad \varphi' \wedge \delta' \prec \delta \\
\hline
 & \multicolumn{2}{c}{\mathcal{C}omp(\mathcal{B}) = \emptyset}
\end{array}$$

Figure 2: Rule WELL.

premises W2 and W3, there exists an infinite sequence of states over which the ranking function δ decreases infinitely many times, and never increases. Since δ is defined over a well-founded domain, this is clearly impossible, contradicting our assumption.

The completeness of rule WELL is stated by the following claim:

Claim 2 *Let $\mathcal{B}: \langle V, \Theta, \rho, \mathcal{J} : \{J\}, \mathcal{C} : \emptyset\rangle$ be a* BDS. *If $\mathcal{B}$ is infeasible, then there exist an assertion φ, a well founded domain $(\mathcal{W}, \prec)$ and a ranking function $\delta: \Sigma_V \mapsto \mathcal{W}$ satisfying the premises of rule* WELL.

Namely, if we can prove state validities (W1, W2 and W3), we can prove that $\mathcal{B}$ is infeasible.

Proof (sketch)**:** To prove the claim, we have to find both an assertion φ and a ranking function δ that satisfy the premises W1–W3 of rule WELL.

The proof of existence of an assertion φ characterizing the set of all reachable states of a BDS is presented in [MP91a] and discussed in more detail in [MP91b] (Section 2.5). The assertion (using predicate calculus) is constructed as an encoding of the finite paths to a reachable state, using the initial condition Θ and the transition relation ρ of $\mathcal{B}$ to constrain the path.

The existence of a well founded domain $(\mathcal{W}, \prec)$ and ranking function δ satisfying the premises W1–W3, is shown in [Var91], based on [LPS81]. The syntactic representation of the well founded predicates $\delta' \preceq \delta$ and $\delta' \prec \delta$, using an assertion language based on the predicate calculus augmented with minimal and maximal fixpoints operators, is discussed in [MP91a] based on [SdRG89].

5 Finitary Abstraction of a BDS

In this section, we present a general methodology for *abstraction* of a BDS, derived from the notion of abstract interpretation [CC77]. For more details see [KP99b]. Let $\mathcal{B}$ =

$\langle V, \Theta, \rho, \mathcal{J}\colon\{J\}, \mathcal{C}\colon\emptyset\rangle$ be a BDS, and Σ denote the set of states of $\mathcal{B}$, the *concrete states*. Let $V_A = \{U_1, \ldots, U_m\}$ be a set of typed variables, to which we refer as the *abstract variables*. An *abstraction presentation* is a list of definitions $\alpha : (U_1 = \mathcal{E}^\alpha_1(V), \ldots, U_m = \mathcal{E}^\alpha_m(V))$, where each $\mathcal{E}^\alpha_i(V)$ is an expression over V. The abstraction presentation α induces a mapping f_α from Σ, the set of V-states, into Σ_A, the set of V_A-states, where $S = f_\alpha(s)$ if, for every $i = 1, \ldots, m$, the value of U_i in S equals the value of $\mathcal{E}^\alpha_i$ in s. When there is no danger of confusion, we refer to α simply as an *abstraction* (or *abstraction mapping*) and write $S = \alpha(s)$ instead of $S = f_\alpha(s)$. We say that α is a *finitary abstraction mapping* if Σ_A is a finite set.

Let $p(V)$ be an assertion. We define the operator α^+, as follows

$$\alpha^+(p(V)):\quad \exists V\ \Big(V_{_A} = \mathcal{E}^\alpha(V) \quad\wedge\quad p(V)\Big).$$

Note that the free variables of the assertion $\alpha^+(p(V))$ are the abstract variables V_A. The assertion $\alpha^+(p)$ holds for an abstract state $S \in \Sigma_{_A}$ iff the assertion p holds for *some* concrete state $s \in \Sigma$ such that $s \in \alpha^{-1}(S)$, i.e., some state s such that $S = \alpha(s)$. Alternatively, $\alpha^+(p)$ is the smallest set $X \subseteq \Sigma_A$ such that $\|p\| \subseteq \alpha^{-1}(X)$, where $\|p\|$ represents the set of states which satisfy the assertion p. For readers who prefer to view abstractions via the framework of Galois connections [CC77], we point out that the pair (α^+, a^{-1}) form a *Galois insertion* [MSS86] between the concrete lattice 2^Σ and the abstract lattice 2^{Σ_A}. That is, for every abstract set $A \subseteq \Sigma_{_A}$ and concrete set $C \subseteq \Sigma$, we have (with some abuse of notation)

$$A = \alpha^+(\alpha^{-1}(A)) \qquad \text{and} \qquad C \subseteq \alpha^{-1}(\alpha^+(C))$$

Let $\mathcal{B} = \langle V, \Theta, \rho, \mathcal{J} = \{J\}, \mathcal{C} = \emptyset\rangle$ be a BDS. We define $\mathcal{B}^\alpha = \langle V_{_A}, \Theta^\alpha, \rho^\alpha, \mathcal{J}^\alpha, \mathcal{C}^\alpha\rangle$, the α-abstracted BDS, as follows:

$$\Theta^\alpha \;=\; \alpha^+(\Theta) \qquad \rho^\alpha \;=\; \alpha^{++}(\rho) \qquad \mathcal{J}^\alpha \;=\; \{\alpha^+(J)\} \qquad \mathcal{C}^\alpha \;=\; \emptyset$$

where for an arbitrary formula $\varphi(U_1, U_2)$, the operator α^{++} is defined by

$$\alpha^{++}(\varphi):\quad \exists U_1, U_2\ (V_{_A} = \mathcal{E}^\alpha(U_1) \ \wedge\ V'_{_A} = \mathcal{E}^\alpha(U_2) \ \wedge\ \varphi(U_1, U_2)).$$

In practice, more efficient ways of computing the abstract transition relation are used, as discussed in [CU98].

The following claim relates the computations of $\mathcal{B}$ to the computations of $\mathcal{B}^\alpha$.

Claim 3 *Let $\sigma = s_0, s_1, \ldots$ be a computation of $\mathcal{B}$. Then the sequence $\sigma_A = S_0, S_1, \ldots,$ where $S_i = \alpha(s_i)$ for every $i \geq 0$, is a computation of $\mathcal{B}^\alpha$.*

Proof: Denote by $U^j = s_j[V]$ and $U^j_A = S_j[V_A]$, the values of the system variables V and V_A in the states s_j and S_j, respectively, for every $j \geq 0$. Since $\sigma \in \mathit{Comp}(\mathcal{B})$, then $U^0 \models \Theta$, namely $\Theta(U^0) = \mathrm{T}$. Since $U^0_A = \mathcal{E}^\alpha(U^0)$, then $\Theta^\alpha(U^0_A) = (U^0_A = \mathcal{E}^\alpha(U^0) \wedge\ \Theta(U^0)) = \mathrm{T}$, namely $\Theta^\alpha(U^0_A) = \mathrm{T}$.

We proceed by considering two successive states, s_j, s_{j+1} in σ. Again, since $\sigma \in \mathit{Comp}(\mathcal{B})$ then $\rho(U^j, U^{j+1}) = \mathrm{T}$. Then,

$$\rho^\alpha(U^j_A, U^{j+1}_A) = (U^J_A = \mathcal{E}^\alpha(U^j) \wedge U^{J+1}_A = \mathcal{E}^\alpha(U^{j+1}) \wedge \rho(U^j, U^{j+1})) = \mathrm{T}.$$

Finally, since σ is a computation of $\mathcal{B}$, there exists infinitely many states in σ satisfying J. Let s_k be such a state, namely $J(U^k) = \text{T}$. Then,$J^\alpha(U_A^k) = (U_A^k = \mathcal{E}^\alpha(U^k) \wedge J(U^k)) = \text{T}$.

∎

The following claim states the soundness of the automata-theoretic approach to verification by finitary abstraction:

Corollary 4 (Weak Preservation) $\quad Comp(\mathcal{B}^\alpha) = \emptyset$ *implies* $Comp(\mathcal{B}) = \emptyset$.

Proof: Immediate result from Claim 3.

∎

As an example, consider program BAKERY-2, presented in Fig. 3, which implements the BAKERY algoritnm for mutual exclusion by Lamport [Lam74].

$$\textbf{local} \quad y_1, y_2 \quad : \textbf{natural where } y_1 = y_2 = 0$$

$$\left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \left[\begin{array}{ll} \ell_1 : & \textbf{NonCritical} \\ \ell_2 : & y_1 := y_2 + 1 \\ \ell_3 : & \textbf{await } y_2 = 0 \ \vee \ y_1 < y_2 \\ \ell_4 : & \textbf{Critical} \\ \ell_5 : & y_1 := 0 \end{array}\right] \end{array}\right] \quad || \quad \left[\begin{array}{l} m_0 : \textbf{loop forever do} \\ \left[\begin{array}{ll} m_1 : & \textbf{NonCritical} \\ m_2 : & y_2 := y_1 + 1 \\ m_3 : & \textbf{await } y_1 = 0 \ \vee \ y_2 \leq y_1 \\ m_4 : & \textbf{Critical} \\ m_5 : & y_2 := 0 \end{array}\right] \end{array}\right]$$

$$- P_1 - \qquad\qquad\qquad\qquad - P_2 -$$

Figure 3: Program BAKERY-2: the Bakery algorithm for two processes.

Program BAKERY-2 is an infinite-state system, since y_1 and y_2 can assume arbitrarily large values. The temporal properties we wish to establish are

$$\psi_{exc} \ : \ \Box \neg(at\_\ell_4 \ \wedge \ at\_m_4) \quad \text{and} \quad \psi_{acc} \ : \ \Box(at\_\ell_2 \ \rightarrow \ \Diamond \, at\_\ell_4).$$

The safety property ψ_{exc} requires *mutual exclusion*, guaranteeing that the two processes never co-reside in their respective critical section at the same time. The liveness property ψ_{acc} requires *accessibility* for process P_1, guaranteeing that, whenever P_1 reaches location ℓ_2 it will eventually reach location ℓ_4. As described in section 3, we construct the testers $T_{\neg\psi_{exc}}$ and $T_{\neg\psi_{acc}}$ representing all the sequences violating ψ_{exc} and ψ_{acc}, respectively. Both testers are finite state.

Following [BBM95], we define abstract Boolean variables $B_{p_1}, B_{p_2}, \ldots, B_{p_k}$, one for each atomic data formula, where the atomic data formulas for BAKERY-2 are $y_1 = 0$, $y_2 = 0$, and $y_1 < y_2$. The abstract system variables consist of the concrete control variables, which are left unchanged, and a set of abstract Boolean variables $B_{p_1}, B_{p_2}, \ldots, B_{p_k}$. The abstraction mapping α is defined by

$$\alpha: \quad \{B_{p_1} = p_1, B_{p_2} = p_2, \ldots, B_{p_k} = p_k\}$$

That is, the Boolean variable B_{p_i} has the value *true* in the abstract state iff the assertion p_i holds at the corresponding concrete state. It is straightforward to compute the α-induced abstractions of the initial condition Θ^α and the transition relation ρ^α. In Fig. 4, we present program BAKERY-2 (with a capital B), the α-induced abstraction of program BAKERY-2.

$$\textbf{local}\quad B_{y_1=0}, B_{y_2=0}, B_{y_1<y_2}\quad : \textbf{Boolean initially } B_{y_1=0} = B_{y_2=0} = 1, B_{y_1<y_2} = 0$$

$$\left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \left[\begin{array}{ll} \ell_1 : & \textbf{NonCritical} \\ \ell_2 : & (B_{y_1=0}, B_{y_1<y_2}) := (0,0) \\ \ell_3 : & \textbf{await } B_{y_2=0} \ \vee \ B_{y_1<y_2} \\ \ell_4 : & \textbf{Critical} \\ \ell_5 : & (B_{y_1=0}, B_{y_1<y_2}) := (1, \neg B_{y_2=0}) \end{array}\right] \end{array}\right] \quad || \quad \left[\begin{array}{l} m_0 : \textbf{loop forever do} \\ \left[\begin{array}{ll} m_1 : & \textbf{NonCritical} \\ m_2 : & (B_{y_2=0}, B_{y_1<y_2}) := (0,1) \\ m_3 : & \textbf{await } B_{y_1=0} \ \vee \ \neg B_{y_1<y_2} \\ m_4 : & \textbf{Critical} \\ m_5 : & (B_{y_2=0}, B_{y_1<y_2}) := (1,0) \end{array}\right] \end{array}\right]$$

$$- P_1 - \qquad\qquad\qquad - P_2 -$$

Figure 4: Program BAKERY-2: the Bakery algorithm for two processes.

Since the properties we wish to verify refer only to the control variables (through the at_ℓ and at_m expressions), both $T_{\neg\psi_{exc}}$ and $T_{\neg\psi_{acc}}$ are not affected by the abstraction. The synchronous compositions $\mathcal{D}_{\text{BAKERY}-2} ||| T_{\neg\psi_{exc}}$ and $\mathcal{D}_{\text{BAKERY}-2} ||| T_{\neg\psi_{acc}}$ are finite-state FDS's whose infeasibility can now be model-checked. By Claims 1 and 4, we can infer that the original program BAKERY-2 satisfies the temporal properties ψ_{exc} and ψ_{acc}.

6 Augmentation by Progress Monitors

Program BAKERY-2 is an example of successful data abstraction. However, there are cases where abstraction alone is inadequate for transforming an infinite-state system satisfying a property into a finite-state abstraction which maintains the property. In the following we first illustrate the problem and the proposed solution, then present the general solution.

6.1 An Illustrative Examples

In Fig. 5, we present a simple looping program. The assignment at statement ℓ_2 assigns to y non-deterministically the values $y+1$ or y. The property we wish to verify is that program SUB-ADD always terminates.
A natural abstraction for the variable y is defined by

$$Y = \textbf{if } y = 0 \textbf{ then } \mathit{zero} \textbf{ else if } y = 1 \textbf{ then } \mathit{one} \textbf{ else } \mathit{large},$$

where y is abstracted into the three-valued domain $\{\mathit{zero}, \mathit{one}, \mathit{large}\}$. Applying this abstraction yields the abstract program SUB-ADD-ABS-1, presented in Fig. 6, where the abstract

$$
\begin{array}{|l|}
\hline
\qquad\qquad y\text{: } \textbf{natural} \\
\ell_0 : \quad \textbf{while } y > 1 \textbf{ do} \\
\qquad\qquad \left[\begin{array}{ll} \ell_1 : & y := y - 2 \\ \ell_2 : & y := \{y+1, y\} \\ \ell_3 : & \textbf{skip} \end{array}\right] \\
\ell_4 : \\
\hline
\end{array}
$$

Figure 5: Program SUB-ADD .

functions *sub2* and *add1* are defined by

$$
\begin{array}{lcl}
\mathit{sub2}(Y) & = & \textbf{if } Y = \{\mathit{zero}, \mathit{one}\} \textbf{ then } \mathit{zero} \textbf{ else } \{\mathit{zero}, \mathit{one}, \mathit{large}\}, \\
\mathit{add1}(Y) & = & \textbf{if } Y = \mathit{zero} \textbf{ then } \mathit{one} \textbf{ else } \mathit{large}.
\end{array}
$$

$$
\begin{array}{|l|}
\hline
\qquad\qquad Y \;\; : \{\mathit{zero}, \mathit{one}, \mathit{large}\} \\
\ell_0 : \quad \textbf{while } Y = \mathit{large} \textbf{ do} \\
\qquad \left[\begin{array}{lrcl} \ell_1 : & Y & := & \mathit{sub2}(Y) \\ \ell_2 : & Y & := & \{\mathit{add1}(Y), Y\} \\ \ell_3 : & \textbf{skip} & & \end{array}\right] \\
\ell_4 : \\
\hline
\end{array}
$$

Figure 6: Program SUB-ADD-ABS-1 abstracting program SUB-ADD.

Unfortunately, program SUB-ADD-ABS-1 need not terminate, because the function *sub2* can always choose to yield *large* as a result. Termination of programs like program SUB-ADD can always be established by identification of a *progress measure* that never increases and sometimes is guaranteed to decrease. In this case, for example, we can use the progress measure $\delta : y + at\_\ell_2$, which never increases and always decreases on the execution of statement ℓ_1.

To obtain a working abstraction, we first compose program SUB-ADD with an additional module, called the *progress monitor* for the measure δ, as shown in Fig. 7.
The construct **always do** appearing in MONITOR M_δ means that the assignment that is the body of this construct is executed at *every* step. The comparison function $\mathit{diff}(\delta, \delta')$ is defined by

$$
\mathit{diff}(\delta, \delta') \;=\; \textbf{if } \delta < \delta' \textbf{ then } 1 \textbf{ else if } \delta = \delta' \textbf{ then } 0 \textbf{ else } -1.
$$

$$
\begin{array}{c}
y\text{: } \textbf{natural} \\[4pt]
\left[\begin{array}{ll}
\ell_0: & \textbf{while } y > 1 \textbf{ do} \\
& \left[\begin{array}{ll}
\ell_1: & y := y - 2 \\
\ell_2: & y := \{y+1, y\} \\
\ell_3: & \textbf{skip}
\end{array}\right] \\
\ell_4: &
\end{array}\right]
\;\Big|\!\Big|\!\Big|\;
\left[\begin{array}{ll}
\textbf{define} & \delta = y + at\_\ell_2 \\
inc & : \{-1, 0, 1\} \\
m_0: & \textbf{always do} \\
& \quad inc := \mathit{diff}(\delta, \delta')
\end{array}\right] \\[4pt]
\text{— SUB – ADD —} \qquad\qquad \text{— MONITOR } M_\delta \text{ —}
\end{array}
$$

Figure 7: Program SUB-ADD composed with a monitor.

The presentation of the monitor module M_δ in Fig. 7 is only for illustration purposes. The precise definition of this module is given by the following FDS:

$$
\begin{array}{lll}
V: \{V_{\mathcal{D}}, inc : \{-1, 0, 1\}\} & \Theta : \mathrm{T} & \\
\rho : inc' = \mathit{diff}(\delta, \delta') & \mathcal{J} : \emptyset & \mathcal{C} : \{(inc < 0, inc > 0)\}
\end{array}
$$

where $V_{\mathcal{D}}$ are the system variables of the FDS representing the program SUB-ADD. Thus, at every step of the computation, module M_δ compares the new value of δ with the current value, and sets variable *inc* to -1, 0, or 1, according to whether the value of δ has decreased, stayed the same, or increased, respectively. This FDS has no justice requirements but has the single compassion requirement $(inc < 0, inc > 0)$ stating that δ cannot decrease infinitely many times without also increasing infinitely many times. This requirement is a direct consequence of the fact that δ ranges over the well-founded domain of the natural numbers, which does not allow an infinitely descreasing sequence.

It is possible to represent this composition as equivalent to the sequential program presented in Fig. 8, where we have conjoined the repeated assignment of module M_δ with every assignment of process SUB-ADD. The abstraction of the program of Fig. 8 abstracts y into

$$
\begin{array}{l}
\begin{array}{rl}
y & : \textbf{natural} \\
inc & : \{-1, 0, 1\}
\end{array} \\[4pt]
\ell_0: \; \textbf{while } y > 0 \textbf{ do} \\
\qquad \left[\begin{array}{lrcr}
\ell_1: & (y, inc) & := & (y-2,\ \mathit{diff}(\delta, \delta')) \\
\ell_2: & (y, inc) & := & (\{y+1, y\},\ \mathit{diff}(\delta, \delta')) \\
\ell_3: & inc & := & \mathit{diff}(\delta, \delta')
\end{array}\right] \\
\ell_4:
\end{array}
$$

Figure 8: A sequential equivalent of the monitored program.

a variable Y ranging over $\{zero, one, large\}$. The variable *inc* is not abstracted. The resulting abstraction is presented in Fig. 9. The program SUB-ADD-ABS-2 (Fig. 9) differs from program SUB-ADD-ABS-1 (Fig 6) by being (synchronously) composed with a progress monitor, which introduces the additional compassion requirement $(inc < 0, inc > 0)$. It is this additional requirement that forces program SUB-ADD-ABS-2 to terminate. This is because a run in which *sub1* always yields *large* as a result is a run in which *inc* is negative infinitely many times (on every visit to ℓ_1) and is never positive beyond the first state. The fact that SUB-ADD-ABS-2 always terminates can now be successfully model checked.

$$
\begin{array}{ll}
 & \begin{array}{ll} Y & : \{zero, one, large\} \\ inc & : \{-1, 0, 1\} \\ \textbf{compassion} & (inc < 0, inc > 0) \end{array} \\
\ell_0 : & \textbf{while } Y = large \textbf{ do} \\
 & \left[\begin{array}{lrcl} \ell_1 : & (Y, inc) & := & (sub2(Y), -1) \\ \ell_2 : & (Y, inc) & := & (\{add1(Y), Y\}, \{0, -1\}) \\ \ell_3 : & inc & := & 0 \end{array} \right] \\
\ell_4 : &
\end{array}
$$

Figure 9: Program SUB-ADD-ABS-2: Abstracted version of the monitored program.

6.2 The General Structure of a Progress Monitor

We proceed to define the general structure of a progress monitor and show that its augmentation to a system being verified is safe. Let $\mathcal{D}$ be an FDS with a set $V_{\mathcal{D}}$ of system variables. Let $(\mathcal{W}, \prec)$ be a well founded domain, and δ be a ranking function for $\mathcal{D}$, mapping the states of $\mathcal{D}$ into the well-founded domain $(\mathcal{W}, \prec)$. A *progress monitor* for δ is an FDS M_δ of the following form:

$$
M_\delta = \left\langle \begin{array}{llllll} V : & \{V_{\mathcal{D}}, inc : \{-1, 0, 1\}\}, & \Theta : & true, & & \\ \rho : & inc' = \mathit{diff}(\delta(V_{\mathcal{D}}), \delta(V'_{\mathcal{D}})), & \mathcal{J} : & \emptyset, & \mathcal{C} : & \{(inc < 0, inc > 0)\} \end{array} \right\rangle
$$

The following claim states the soundness of the augmentation by a progress monitor:

Claim 5 (Soundness of Augmentation) $\quad Comp(\mathcal{D} ||| M_\delta) \Downarrow_{V_{\mathcal{D}}} = Comp(\mathcal{D})$

Proof: The transition relation of the progress monitor M_δ affects only the variable *inc*, which is a variable not in $V_{\mathcal{D}}$.

∎

7 Verification by Augmented Finitary Abstraction

Let $\mathcal{D}$ be an FDS and $T_{\neg\psi}$ be a BDS representing some (negated) property over $\mathcal{D}$, such that $Comp(\mathcal{D}) \cap Comp(T_{\neg\psi}) = \emptyset$. We can now formulate the automata-theoretic approach to *verification by augmented finitary abstraction* (VAA) as follows.

To verify that $Comp(\mathcal{D}) \cap Comp(T_{\neg\psi}) = \emptyset$,

- Construct the synchronous parallel composition $\mathcal{D}|||T_{\neg\psi}$ of the FDS $\mathcal{D}$ representing the system to be verified and the BDS $T_{\neg\psi}$ representing the (negated) property, and transform it into an equivalent BDS $\mathcal{B}_{(\mathcal{D},\neg\psi)}$.
- Identify an appropriate ranking function Δ for $\mathcal{B}_{(\mathcal{D},\neg\psi)}$, and construct the progress monitor M_Δ.
- Construct an FDS of the augmented system $\mathcal{A}: \quad \mathcal{B}_{(\mathcal{D},\neg\psi)}|||M_\Delta$.
- Identify an appropriate abstraction function α.
- Abstract the augmented system $\mathcal{A}$ into a *finitary abstract* FDS $\mathcal{A}^\alpha$.
- Model-check $Comp(\mathcal{A}^\alpha) = \emptyset$.
- Infer $Comp(\mathcal{D}) \cap Comp(T_{\neg\psi}) = \emptyset$.

Claim 6 $\quad Comp((\mathcal{B}_{(\mathcal{D},\neg\psi)}|||M_\Delta)^\alpha) = \emptyset \quad \textit{implies} \quad Comp(\mathcal{D}) \cap Comp(T_{\neg\psi}) = \emptyset$

Proof: Assume that $Comp((\mathcal{B}_{(\mathcal{D},\neg\psi)}|||M_\Delta)^\alpha) = \emptyset$. Then by Corollary 4 and Claim 5, we obtain $Comp(\mathcal{D}) \cap Comp(T_{\neg\psi}) = \emptyset$.

8 Completeness of the VAA Method

In the following we prove the completeness of the VAA method. First we introduce the operator α^-, and establish some useful properties of the abstraction mappings α^+ and α^{++}.

8.1 The α^- Operator

Let $p(V)$ be an assertion. The operator α^- is defined by

$$\alpha^-(p(V)): \quad map(V_A) \wedge \forall V \left(V_A = \mathcal{E}^\alpha(V) \quad \rightarrow \quad p(V)\right).$$

where $map(V_A)$: $\exists V\left(V_A = \mathcal{E}^\alpha(V)\right)$. The assertion $\alpha^-(p(V))$, like $\alpha^+(P(V))$, has V_A as free variables. The assertion $map(V_A)$ states that V_A is the image of at least one concrete state $s \in \Sigma$. The assertion $\alpha^-(p)$ holds for an abstract state $S \in \Sigma_A$ iff $map(V_A)$ holds and the assertion p holds for *all* concrete states $s \in \Sigma$ such that $s \in \alpha^{-1}(S)$. Thus, when $map(V_A)$ holds, $\alpha^-(p)$ is the largest set of states $X \subseteq \Sigma_A$ such that $\alpha^{-1}(X) \subseteq \|p\|$, where $\|p\|$ represents the set of states that satisfy the assertion p. If $\alpha^-(p)$ is valid, then $\|\alpha^-(p)\| = \Sigma_A$ implying $\alpha^{-1}(\|\alpha^-(p)\|) = \Sigma$ which, by the above inclusion, leads to $\|p\| = \Sigma$ establishing the validity of p.

Note the duality relations holding between α^+ and α^-, which can be expressed by the equivalences

$$\neg\alpha^+(p) \;\sim\; map(V_A) \rightarrow \alpha^-(\neg p) \quad (1)$$
$$\neg\alpha^-(p) \;\sim\; map(V_A) \rightarrow \alpha^+(\neg p) \quad (2)$$

or, equivalently, by

$$\alpha^+(\neg p) \;\sim\; \neg\alpha^-(p) \;\wedge\; map(V_A) \quad (3)$$
$$\alpha^-(\neg p) \;\sim\; \neg\alpha^+(p) \;\wedge\; map(V_A) \quad (4)$$

An abstraction α is said to be *precise with respect to an assertion* p if $\alpha^+(p) \sim \alpha^-(p)$. For such cases, we will sometimes write $\alpha^+(p)$ simply as $\alpha(p)$. The following Claim asserts a sufficient condition for α to be precise with respect to an assertion p. The Claim and its proof are presented in [KP99b], and are given here for completeness of the presentation.

Claim 7 (Existence of precise abstractions) *Let* $\alpha = \{U_1 = \mathcal{E}_1(V), \ldots, U_m = \mathcal{E}_m(V), B_p = p(V)\}$ *be a presentation of an abstraction from* V *to* $V_A = \{U_1, \ldots, U_m, B_p\}$. *Then* α *is precise with respect to* $p(V)$.

Proof: The first direction $\alpha^-(p) \rightarrow \alpha^+(p)$ is valid for any assertion p and abstraction α, with no precision requirement. The proof is trivial and is thus omitted.

Next, we prove the second direction $\alpha^+(p) \rightarrow \alpha^-(p)$. Expanding the definitions, we get

$$\exists V : V_A = \mathcal{E}^\alpha(V) \wedge p(V) \quad \rightarrow \quad \forall V : V_A = \mathcal{E}^\alpha(V) \rightarrow p(V) \wedge \exists V : V_A = \mathcal{E}^\alpha(V)$$

Since $\exists V : V_A = \mathcal{E}^\alpha(V) \wedge p(V)$ implies $\exists V : V_A = \mathcal{E}^\alpha(V)$, we only have to show

$$\exists V : V_A = \mathcal{E}^\alpha(V) \wedge p(V) \quad \rightarrow \quad \forall V : V_A = \mathcal{E}^\alpha(V) \rightarrow p(V)$$

We split V_A into $U_A \cup \{B_p\}$, expanding $V_A = \mathcal{E}^\alpha(V)$ to

$$U_A = \mathcal{E}^\alpha_U(V) \wedge B_p = p(V).$$

Assuming the antecedent, and skolemizing the existential quantifier in the antecedent by v_1, we get

$$U_A = \mathcal{E}^\alpha_U(v_1) \wedge B_p = p(v_1) \wedge p(v_1).$$

It follows that $B_p = \mathrm{T}$. Thus, for every value v_2, if

$$\left(U_A = \mathcal{E}^\alpha_U(v_2) \wedge B_p = p(v_2)\right)$$

holds, then $p(v_2)$ must hold. ∎

8.2 Properties of α^+ and α^{++}

The following lemma states a basic property of α^+.

Lemma 8 *If α is precise with respect to either p or q, then*

$$\alpha^+(p \ \wedge \ q) \quad \sim \quad \alpha^+(p) \ \wedge \ \alpha^+(q)$$

Proof: For the general case (i.e., no preciseness constraints) we can only claim that

$$\alpha^+(p \ \wedge \ q) \quad \text{implies} \quad \alpha^+(p) \ \wedge \ \alpha^+(q).$$

For the special case that α is precise with respect to q (i.e., $\alpha^+(q) \sim \alpha^-(q)$), we do have the equivalence

$$\alpha^+(p \ \wedge \ q) \quad \sim \quad \alpha^+(p) \ \wedge \ \alpha^+(q).$$

To see this, it is only necessary to establish that $\alpha^+(p) \ \wedge \ \alpha^+(q)$ implies $\alpha^+(p \ \wedge \ q)$. This is established by the following chain of equivalences/implications, assuming preciseness with respect to q:

$$\begin{array}{l}
\alpha^+(p) \ \wedge \ \alpha^+(q) \quad \sim \quad \alpha^+(p) \ \wedge \ \alpha^-(q) \quad \sim \\
\exists V : (V_A = \mathcal{E}^\alpha(V) \ \wedge \ p(V)) \quad \wedge \quad \forall V : (V_A = \mathcal{E}^\alpha(V) \ \rightarrow \ q(V)) \qquad \text{implies} \\
\exists V : V_A = \mathcal{E}^\alpha(V) \ \wedge \ (p(V) \ \wedge \ q(V)) \quad \sim \quad \alpha^+(p \ \wedge \ q)
\end{array}$$

By symmetry, $\alpha^+(p \ \wedge \ q) \sim (\alpha^+(p) \ \wedge \ \alpha^+(q))$ also for the case that α is precise with respect to p.

∎

The following lemma states some of the properties of α^{++}.

Lemma 9 *If α is precise with respect to $r(V)$, then*

$$\alpha^{++}(p \ \wedge \ r) \ \sim \ \alpha^{++}(p) \ \wedge \ \alpha^+(r) \tag{5}$$

$$\alpha^{++}(p \ \wedge \ r') \ \sim \ \alpha^{++}(p) \ \wedge \ (\alpha^+(r))' \tag{6}$$

where $(\alpha^+(r))'$ is defined by $\exists V : V'_A = \mathcal{E}^\alpha(V) \ \wedge \ r(V)$, and $\alpha^{++}(p \ \wedge \ r')$ is given by $\exists U_1, U_2 \, (V_A = \mathcal{E}^\alpha(U_1) \ \wedge \ V'_A = \mathcal{E}^\alpha(U_2) \ \wedge \ p(U_1, U_2) \ \wedge \ r(U_2))$.

Proof of equation (5): The first direction

$$\alpha^{++}(p \ \wedge \ r) \quad \text{implies} \quad \alpha^{++}(p) \ \wedge \ \alpha^+(r).$$

is obvious, and does not require any precision constraints. For the special case that α is precise with respect to r (i.e., $\alpha^+(r) \sim \alpha^-(r)$), we do have the equivalence

$$\alpha^{++}(p \ \wedge \ r) \quad \sim \quad \alpha^{++}(p) \ \wedge \ \alpha^+(r). \tag{7}$$

To see this, it is only necessary to establish that $\alpha^{++}(p) \ \wedge \ \alpha^+(r)$ implies $\alpha^{++}(p \ \wedge \ r)$. This is established by the following chain of equivalences/implications:

$$\begin{array}{l}
\alpha^{++}(p) \ \wedge \ \alpha^+(r) \quad \sim \quad \alpha^{++}(p) \ \wedge \ \alpha^-(r) \quad \sim \\
\exists U_1, U_2 : (V_A = \mathcal{E}^\alpha(U_1) \ \wedge \ V'_A = \mathcal{E}^\alpha(U_2) \ \wedge \ p(U_1, U_2)) \quad \wedge \quad \forall V : (V_A = \mathcal{E}^\alpha(V) \ \rightarrow \ r(V))
\end{array}$$

which implies

$$\exists U_1, U_2 : V_A = \mathcal{E}^\alpha(U_1) \;\wedge\; V_A' = \mathcal{E}^\alpha(U_2) \;\wedge\; (p(U_1, U_2) \;\wedge\; r(U_1)) \quad \sim \quad \alpha^{++}(p \;\wedge\; r)$$

▪

The proof of equation (6) is similar.
It follows from the definitions that if $p = p(V)$, then both $\alpha^{++}(p) \sim \alpha^+(p)$ and $\alpha^{++}(p') \sim (\alpha^+(p))'$ hold without any precision assumptions about p.
Finally, we observe that if an implication is valid, we can apply the abstractions α^+ and α^{++} to both sides of the implication.

Lemma 10 $\models p \;\rightarrow\; q$ implies $\left(\begin{array}{lccll} \models & \alpha^+(p) & \rightarrow & \alpha^+(q) & \text{and} \\ \models & \alpha^{++}(p) & \rightarrow & \alpha^{++}(q) & \end{array}\right)$

Proof: Assume $\models p \;\rightarrow\; q$. Suppose $\alpha^+(p)$ holds in an abstract state. Then this state is an image of a concrete state satisfying p, and consequently also q. It follows that $\alpha^+(q)$ also holds in the abstract state. The argument for $\models \alpha^{++}(p) \rightarrow \alpha^{++}(q)$ is analogous. ▪

8.3 The Completeness Statement

Let $\mathcal{B}$ be an infeasible BDS. Let α be an abstraction mapping and δ be a ranking function for $\mathcal{B}$. We say that $\langle \alpha, \delta \rangle$ is an *adequate augmented abstraction* for $\mathcal{B}$ if α is finitary and $Comp((\mathcal{B} ||| M_\delta)^\alpha) = \emptyset$.

Claim 11 (Completeness of VAA) *Let $\mathcal{B}_{(\mathcal{D},\neg\psi)}$ be an infeasible* BDS*. Then, there exists an adequate augmented abstraction for $\mathcal{B}_{(\mathcal{D},\neg\psi)}$.*

Based on Claim 2, there exists an assertion and a ranking function that satisfy the three premises of rule WELL (section 4) for the BDS $\mathcal{B}_{(\mathcal{D},\neg\psi)}$. We denote these assertion and ranking function by Φ and Δ, respectively. We choose α to be an arbitrary finitary abstraction that is precise with respect to the assertions Θ (the initial condition of $\mathcal{B}_{(\mathcal{D},\neg\psi)}$), Φ and the single J of the BDS $\mathcal{B}_{(\mathcal{D},\neg\psi)}$. Namely, based on Claim 7, we choose any finitary abstraction and augment it with the three Boolean variables B_θ, B_Φ and B_J defined by $B_\theta = \Theta$, $B_\Phi = \Phi$ and $B_J = J$.

We require that α does not abstract the auxiliary variable *inc*. Let $\mathcal{A} = \mathcal{B}_{(\mathcal{D},\neg\psi)} ||| M_\Delta$. In the following, we show that for this choice of ranking function and abstraction mapping, $Comp(\mathcal{A}^\alpha) = \emptyset$, that is, $\langle \alpha, \Delta \rangle$ is an *adequate augmented abstraction* for $\mathcal{B}_{(\mathcal{D},\neg\psi)}$.

Abstracting the Premises of Rule WELL.

The proof is based on the abstraction of premises W1–W3 of rule WELL , applied to the BDS $\mathcal{B}_{(\mathcal{D},\neg\psi)}$: $\langle V, \Theta, \rho, \mathcal{J} = \{J\}, \mathcal{C} = \emptyset \rangle$. These three premises are known to be valid for our choice of Φ and Δ. Recall that $\mathcal{A}$ is the FDS $\mathcal{B}_{(\mathcal{D},\neg\psi)} ||| M_\Delta$. From the definition of M_Δ, the components of $\mathcal{A}$ are given by

$$\Theta_\mathcal{A} : \;\; \Theta \qquad \rho_\mathcal{A} : \;\; \rho \wedge \underbrace{inc' = \mathit{diff}(\Delta, \Delta')}_{\rho_{M_\Delta}} \qquad \mathcal{J}_\mathcal{A} : \;\; \mathcal{J} \qquad \mathcal{C}_\mathcal{A} : \;\; \{(inc < 0, inc > 0)\}$$

From the implication

$$inc' = \mathit{diff}(\Delta, \Delta') \quad \rightarrow \quad (\Delta' \preceq \Delta \rightarrow inc' \leq 0 \quad \wedge \quad \Delta' \prec \Delta \rightarrow inc' < 0)$$

and the three premises of rule WELL applied to $\mathcal{B}_{(\mathcal{D},\neg\psi)}$, we can obtain the following three valid implications:

$$\begin{array}{lllll}
\text{U1.} & \Theta_{\mathcal{A}} & \rightarrow & \Phi \\
\text{U2.} & \rho_{\mathcal{A}} \wedge \Phi & \rightarrow & \Phi' \wedge inc' \leq 0 \\
\text{U3.} & \rho_{\mathcal{A}} \wedge \Phi \wedge J' & \rightarrow & \Phi' \wedge inc' < 0.
\end{array}$$

Based on Lemma 10, we can apply α^+ to both sides of U1 and apply α^{++} to both sides of U2 and U3. We then simplify the right-hand sides, using the fact that $\alpha^{++}(p') \sim (\alpha^+(p))'$, and that α does not abstract inc. Next, since α is precise with respect to the assertions Φ and J, we use Lemma 9 in order to distribute the abstraction over the conjunctions on the left-hand sides of $U2$ and $U3$. These transformations and simplifications lead to the following three valid abstract implications:

$$\begin{array}{llll}
\text{V1.} & \alpha^+(\Theta_{\mathcal{A}}) & \rightarrow & \alpha^+(\Phi) \\
\text{V2.} & \alpha^{++}(\rho_{\mathcal{A}}) \wedge \alpha^+(\Phi) & \rightarrow & (\alpha^+(\Phi))' \wedge inc' \leq 0 \\
\text{V3.} & \alpha^{++}(\rho_{\mathcal{A}}) \wedge \alpha^+(\Phi) \wedge (\alpha^+(J))' & \rightarrow & (\alpha^+(\Phi))' \wedge inc' < 0.
\end{array}$$

The Augmented System $\mathcal{A}^\alpha$ Has No Computations

We proceed to show that $\mathcal{A}^\alpha$ has no computations ($Comp(\mathcal{A}^\alpha) = \emptyset$). Assume to the contrary, that there exists $\sigma: s_0, s_1, \ldots,$ a computation of $\mathcal{A}^\alpha$.

First we use the implications V1–V3 to show that the assertion $\alpha(\Phi)$ is an invariant of σ. Since σ is a computation of $\mathcal{A}^\alpha$, the first state of σ satisfies $\alpha(\Theta_{\mathcal{A}})$ and we conclude by V1 that the first state of σ satisfies $\alpha(\Phi)$. Proceeding from each state s_j of σ to its successor s_{j+1}, which must be an $\alpha^{++}(\rho_{\mathcal{A}})$-successor of s_j, we see from V2 and V3 that $\alpha(\Phi)$ keeps propagating. It follows that $\alpha(\Phi)$ is an invariant of σ, i.e, every state s_i of σ satisfies $\alpha(\Phi)$.

Next, since σ is a computation of $\mathcal{A}^\alpha$, it must contain infinitely many states that satisfy $\alpha(J)$. According to implications V2 and V3, the variable inc is never positive, and is negative infinitely many times. Such a behavior contradicts the compassion requirement $(inc < 0, inc > 0)$ associated with $\mathcal{A}^\alpha$. Thus, σ cannot be a computation of $\mathcal{A}^\alpha$, contradicting our initial assumption. This concludes our proof of completeness.

9 Conclusions

We have presented a method for verification by augmented finitary abstraction by which, in order to verify that a (potentially infinite-state) system satisfies a temporal property, one proves the infeasibility of a finite-state system. The finite-state system is obtained by abstracting a system that is obtained by taking the synchronous composition of the original system, a system that expresses the negation of the temporal property, and a non-constraining progress monitor.

The method has been shown to be sound and complete. It is interesting to note that since the completeness proof only requires the abstraction to be precise with respect to three assertions, the finite-state system that is the result of the abstraction need not contain more than eight states. One is reminded of what is known as *Hoare's Law of Large Programs*: "Inside every large program is a small program struggling to get out".

In principle, the established completeness promotes the VAA method to the status of a viable alternative to the verification of infinite-state reactive systems by temporal deduction. Some potential users of formal verification may find the activity of devising good abstraction mappings more tractable (and similar to programming) than the design of auxiliary invariants. Of course, on a deeper level, it is possible to argue that this is only a formal shift and that the same amount of ingenuity and deep understanding of the analyzed system is still required for effective verification via abstraction as in the practice of temporal deduction methods.

The development of the VAA theory calls for additional research in the implementation of these methods. In particular, there is a strong need for devising heuristics for the automatic generation of effective abstraction mappings and corresponding augmenting monitors.

References

[BBM95] N. Bjørner, I.A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. In 1st *Intl. Conf. on Principles and Practice of Constraint Programming*, volume 976 of *Lect. Notes in Comp. Sci.*, pages 589–623. Springer-Verlag, 1995.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BMS95] I.A. Browne, Z. Manna, and H.B. Sipma. Generalized Temporal Verification Diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *Lect. Notes in Comp. Sci.*, pages 484–498, December 1995.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual Symposium on Principles of Programming Languages*. ACM Press, 1977.

[CGH97] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1), 1997.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Trans. Prog. Lang. Sys.*, 16(5):1512–1542, 1994.

[CGL96] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking. In *Model Checking, Abstraction and Composition*, volume 152 of *Nato ASI Series F*, pages 477–498. Springer-Verlag, 1996.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. 5th ACM Symp. Princ. of Prog. Lang.*, pages 84–96, 1978.

[Cho74] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *J. Comp. Sys. Sci.*, 8:117–141, 1974.

[CU98] M. A. Colon and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In A.J. Hu and M.Y. Vardi, editors, *Proc. 9^{th} Intl. Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lect. Notes in Comp. Sci.*, pages 293–304. Springer-Verlag, 1998.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Prog. Lang. Sys.*, 19(2), 1997.

[Eme85] E.A. Emerson. Automata, tableaux and temporal logics. In *Proc. Conf. Logics of Programs*, pages 79–88. Lec. Notes in Comp. Sci. 193, Springer-Verlag, 1985.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[KP98] Y. Kesten and A. Pnueli. Modularization and abstraction: The keys to formal verification. In L. Brim, J. Gruska, and J. Zlatuska, editors, *The 23rd International Symposium on Mathematical Foundations of Computer Science*, volume 1450 of *Lect. Notes in Comp. Sci.*, pages 54–71. Springer-Verlag, 1998.

[KP99a] Y. Kesten and A. Pnueli. Verifying liveness by augmented abstraction. In *Annual Conference of the European Association for Computer Science Logic (CSL'99)*, Lect. Notes in Comp. Sci. Springer-Verlag, 1999.

[KP99b] Yonit Kesten and Amir Pnueli. Verification by augmented finitary abstraction. *Inf. and Comp.*, 1999. A special issue. To appear.

[KPR98] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proc. 25th Int. Colloq. Aut. Lang. Prog.*, volume 1443 of *Lect. Notes in Comp. Sci.*, pages 1–16. Springer-Verlag, 1998.

[Kur95] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes.* Princeton University Press, Princeton, New Jersey, 1995.

[Lam74] L. Lamport. A new solution to Dijkstra's concurrent programming problem. *Comm. ACM*, 17(8):453–455, 1974.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[LP84] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 11th ACM Symp. Princ. of Prog. Lang.*, pages 97–107, 1984.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. Princ. of Prog. Lang.*, pages 97–107, 1985.

[LPS81] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *Proc. 8th Int. Colloq. Aut. Lang. Prog.*, pages 264–277. Lec. Notes in Comp. Sci. 115, Springer-Verlag, 1981.

[MAB+94a] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Brown, E. chang, M. Colon, L. DeAlfaro, H. Devarajan, H. Sipma, and T. Uribe. Step: The Stanford Temporal Prover. Technical report, Dept. of Computer Science, Stanford University, 1994.

[MAB+94b] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. De Alfaro, H. Devarajan, H. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Dept. of Comp. Sci., Stanford University, Stanford, California, 1994.

[MBSU98] Z. Manna, A. Brown, H. B. Sipma, and T. E. Uribe. Visual abstractions for temporal verification. In *AMAST'98*. Lect. Notes in Comp. Sci., 1998.

[MP91a] Z. Manna and A. Pnueli. Completing the temporal picture. *Theor. Comp. Sci.*, 83(1):97–130, 1991.

[MP91b] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer Verlag, New York, 1991.

[MP94] Z. Manna and A. Pnueli. Temporal verification diagrams. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lect. Notes in Comp. Sci.*, pages 726–765. Springer-Verlag, 1994.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[MSS86] A.C. Melton, D.A. Schmidt, and D.E. Strecker. Galois connections and computer science applications. In David Pitt, Samson Abramsky, Axel Poigne, and David Rydeheard, editors, *Category Theory and Computer programming*, volume 240 of *Lect. Notes in Comp. Sci.*, pages 299–312. Springer-Verlag, 1986.

[SdRG89] F.A. Stomp, W.-P. de Roever, and R.T. Gerth. The μ-calculus as an assertion language for fairness arguments. *Inf. and Comp.*, 82:278–322, 1989.

[Sis89] A. P. Sistla. On verifying that a concurrent program satisfies a nondeterministic specification. *Information Processing Letters*, 32:17–23, 1989.

[SUM96] H.B. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. In *Proc. 8^{th} Intl. Conference on Computer Aided Verification (CAV'96)*, Lect. Notes in Comp. Sci. Springer-Verlag, July 1996.

[SUM99] H.B. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. *Formal Methods in System Design*, 15(1):49–74, 1999.

[Var91] M. Y. Vardi. Verification of concurrent programs – the automata-theoretic framework. *Annals of Pure Applied Logic*, 51:79–98, 1991.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First IEEE Symp. Logic in Comp. Sci.*, pages 332–344, 1986.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Inf. and Cont.*, 115(1):1–37, 1994.

[Wol83] P. Wolper. Temporal logic can be more expressive. *Inf. and Cont.*, 56:72–99, 1983.