Proving Liveness by Backwards Reachability*

Parosh Aziz Abdulla, Bengt Jonsson, Ahmed Rezine, and Mayank Saksena

Dept. of Information Technology, P.O. Box 337, S-751 05 Uppsala, Sweden
{parosh,bengt,rahmed,mayanks}@it.uu.se

Abstract. We present a new method for proving liveness and termination properties for fair concurrent programs, which does not rely on finding a ranking function or on computing the transitive closure of the transition relation. The set of states from which termination or some liveness property is guaranteed is computed by a backwards reachability analysis. A central technique for handling concurrency is a check for certain commutativity properties. The method is not complete. However, it can be seen as a complement to other methods for proving termination, in that it transforms a termination problem into a simpler one with a larger set of terminated states. We show the usefulness of our method by applying it to existing programs from the literature. We have also implemented it in the framework of Regular Model Checking, and used it to automatically verify non-starvation for parameterized algorithms.

1 Introduction

The last decade has witnessed impressive progress in the ability of tools to verify properties of hardware and software systems (e.g., [9, 15, 23]). The success has to a large extent concerned safety properties, e.g., absence of run-time errors, deadlocks, race conditions, etc. Progress in verification of liveness properties has been less prominent. A major reason is that they are harder to verify than safety properties. For finite-state systems and some parameterized systems, safety properties can be verified by computing (some approximation of) the set of reachable states. Verifying liveness properties, requires at least a repeated search through the state space in enumerative model checkers [23]. In symbolic model checkers, a natural but more expensive technique is to compute the transitive closure of the transition relation. Multiple fairness requirements can make the situation even more complicated. For general infinite-state systems, the difference between safety and liveness properties is even larger. For some classes of systems, such as lossy channel systems, checking safety properties is decidable [5], whereas checking liveness properties is undecidable [4].

The general approach for proving liveness involves finding auxiliary assertions associated with well-founded ranking functions and helpful directions (e.g., [25]). Finding such ranking functions is not easy, and automation requires techniques adapted to specific data domains. Techniques have been developed for programs with integers or reals [11–13, 17, 18], functional programs, [24], and parameterized systems [21, 22].

* This work was supported in part by the Swedish Research Council (http://www.vr.se/)

The main technique of software model checking, using finite-state abstractions [15] has been difficult to apply when proving liveness properties, since abstractions may introduce spurious loops [32] that do not preserve liveness. Podelski and Rybalchenko therefore extended the framework of predicate abstraction to that of *transition predicate abstraction* [31], which involves constructing an abstraction of the transition relation and its transitive closure. However, the transitive closure is harder to compute or approximate than the set of reachable states.

Here, we present a new method for proving liveness using simple reachability analysis, which uses neither computation of transitive closure nor explicit construction of ranking functions. The method assumes that the liveness property has been transformed to the property of termination for a system; which is standard for many classes of liveness properties, including the so-called progress properties (of form $\Box(P \implies \Diamond Q)$). Termination is then checked by backwards reachability analysis, which computes the set of states that are backwards reachable from the set of terminated states under a particular transition relation, which we call a *convergence relation*. Computing the set of backwards reachable states is conceptually easier than finding ranking functions or computing the transitive closure. Thus, liveness properties can be established for a class of systems, provided that there is a powerful way to compute sets of backwards reachable states. For many classes of parameterized and infinite-state systems, the set of backwards reachable states is computable (e.g., [5, 2]). For other classes of infinite-state systems, powerful acceleration techniques have been developed that compute or under-approximate the set of reachable states (e.g., [35, 3]). It should be possible to develop equally powerful techniques for backwards reachability analysis, and apply them to proving liveness properties.

For a simple deterministic (non-concurrent) program, the set of states in which termination is guaranteed can be calculated as the set of states that are backwards reachable from some terminated state. We generalize this observation to develop techniques for using backwards reachability analysis to prove termination for general concurrent programs with arbitrary (weak) fairness (aka justice) requirements; backwards reachability analysis should be the only non-trivial computation on the verified program. A central new technique for handling concurrency is the use of commutativity properties between different actions of the program.

Our technique is in general not complete. It computes an under-approximation of the set of states from which termination is guaranteed. If this under-approximation does not include the states for which one intends to prove termination, there are several ways to increase the power of the method. One way is to repeat the backwards reachability analysis, letting the computed under-approximation play the role of terminated states. One then exploits the fact that our convergence relation increases when the set of terminated states increases: a repeated reachability analysis will therefore improve the under-approximation. Another way is to apply other techniques (e.g. based on ranks or transitive closure computation) to prove termination for the remaining states of interest. Here, we present such a complementary technique, developed particularly for parameterized systems.

To show the usefulness of our method, we apply it to several examples. The first is a program also considered by Podelski and Rybalchenko [31]; our method also handles the other programs in [31]. The second example is the well-known *alternating bit* protocol. This is an example of a lossy channel system, for which liveness properties are undecidable [4]. Our example shows that backwards reachability analysis (which is guaranteed to terminate [5]) can prove liveness properties for some of these systems, although in general they are undecidable. Finally, we have implemented our technique in the framework of regular model checking [7]. We prove starvation-freedom for several parameterized mutual exclusion protocols; some of which we have previously not been able to prove starvation-freedom for using transitive closure computation [6].

Related Work For infinite-state systems, fair termination is typically proven by finding auxiliary assertions associated with well-founded ranking functions and helpful directions (e.g., [25, 26]). Automated construction of such ranking functions is a challenging task, which requires techniques adapted to specific data domains. Recently, significant progress has been achieved for programs that operate on numerical domains, integers or reals [11–13, 17, 18, 20]. Rather few papers present efficient techniques to prove termination for programs that operate on arbitrary data domains. For families of parameterized systems, where each system instance is finite-state, liveness can in principle be proven from the transitive closure, but computation of transitive closure is typically expensive [29]. Another approach is to develop heuristics to automate the search for rank functions [21, 22] and procedures to check the conditions in a general proof rule [26] automatically. A third approach has been to find specialized abstractions, e.g., into integers, which work in certain cases [30].

Podelski and Rybalchenko extend the framework of predicate abstraction to that of *transition predicate abstraction* [31, 32, 28, 19], which can be applied on arbitrary programs. The transitive closure of the transition relation is harder to compute or approximate than the set of reachable states. Extensions of predicate abstraction techniques for synthesizing ranking functions have also been developed by Balaban, Pnueli, and Zuck [8].

Our use of commutativity between actions is inspired by the use of commutativity in partial-order techniques to optimize state-space exploration [16] in finite-state model checking.

Organization of the paper Section 2 contains basic definitions, Section 3 an informal overview of our method, and Section 4 the formal presentation of the method. In Section 5, we verify an example also considered by Podelski and Rybalchenko [31], and the alternating bit protocol. In Section 6, we give experimental results on non-starvation for parameterized mutual exclusion algorithms, and describe our complementary termination rule, particularly developed for parameterized systems. Section 7 contains conclusions.

2 Preliminaries

Programs We consider fair concurrent programs modeled as transition systems. A program may contain a set of actions with (weak) fairness requirements (aka justice), as in, e.g., UNITY [14].

Formally, a *program* $\mathcal{P}$ is a triple $\langle S, \longrightarrow, \mathcal{A} \rangle$, where

- S is a set of *states*,
- $\longrightarrow \subseteq S \times S$ is a *transition relation* on S. We require that the identity relation is included in $\longrightarrow$.
- $\mathcal{A}$ is a finite or countable set of *fair actions*, each of which is a subset of $\longrightarrow$, and required to be deterministic.

An *action* is any subset of the transition relation. We write $s \longrightarrow s'$ for $(s, s') \in \longrightarrow$. For an action α, we use $s \xrightarrow{\alpha} s'$ to denote $(s, s') \in \alpha$. An action α is *enabled* in a state s if there is some state s' such that $s \xrightarrow{\alpha} s'$. The set of states in which the action α is enabled is denoted $En(\alpha)$. If T is a set of states, then $\alpha \wedge T$ denotes the set of pairs (s, s') of states such that $s \xrightarrow{\alpha} s'$ and $s \in T$. For a set $\mathcal{B}$ of actions, let $\mathcal{B} \wedge T$ denote $\{\alpha \wedge T \mid \alpha \in \mathcal{B}\}$. A *computation* of $\mathcal{P}$ from a state $s \in S$ is an infinite sequence of states $s_0\ s_1\ s_2\ \ldots$ such that (i) $s = s_0$; (ii) $s_i \longrightarrow s_{i+1}$ for each $i \geq 0$; and (iii) for each fair action $\alpha \in \mathcal{A}$, there are infinitely many $i \geq 0$ where either $s_i \xrightarrow{\alpha} s_{i+1}$ or $s_i \notin En(\alpha)$.

For a set T of states and action α, let $Pre(\alpha, T)$ be the set of states s such that $s \xrightarrow{\alpha} t$ for some $t \in T$. For a set of actions $\mathcal{B}$, let $Pre^*(\mathcal{B}, T)$ be the union of T and the set of states s such that $s \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} t$ for some $t \in T$ and $\alpha_1, \ldots, \alpha_n \in \mathcal{B}$.

Termination Let $\mathcal{P}$ be a program $\langle S, \longrightarrow, \mathcal{A} \rangle$ and $F \subseteq S$ be a set of *terminated* states. We assume F to be *stable*, i.e., that $s \in F$ and $s \longrightarrow s'$ implies $s' \in F$. Define $\Diamond F$ as the set of states s such that any computation of $\mathcal{P}$ from s contains a state in F. In other words, $\Diamond F$ is the set of states from which termination is guaranteed, in the sense that each computation from s will eventually reach F. In this paper we present methods for computing (an under-approximation of) $\Diamond F$.

We can also consider many classes of liveness properties, e.g., progress properties (of form $\Box(P \implies \Diamond Q)$), by first transforming them to termination properties. There exist standard techniques for such reductions. For example, a program satisfies $\Box(P \implies \Diamond Q)$ if $\Diamond Q$ includes states that can be reached from an initial state in a sequence of transitions that visit P, but have not yet visited Q.

Remarks The restriction that each fair action be deterministic can often be circumvented by representing a nondeterministic action as a union of several deterministic ones. Our definition of program does not mention initial states. When initial states are given, a typical use of our techniques will be to first compute the set of reachable states (or an over-approximation), and let them be the states of the program as defined above.

3 Overview of the Proof Method

In this section, we give an overview of our method for computing a (good) under-approximation of the set $\Diamond F$, where F is a set of states of a program $\mathcal{P} = \langle S, \longrightarrow, \mathcal{A} \rangle$. The inspiration for our method is the simple observation that when $\mathcal{P}$ is a deterministic program with only one fair action α, then $\Diamond F$ is the set $Pre^*(\alpha, F)$. Our goal is therefore a technique for proving termination and liveness properties, where the only non-trivial computation is a predecessor calculation, i.e., computing $Pre^*(\mathcal{B}, T)$ for some set of states T and actions $\mathcal{B}$.

Our method works by computing a so-called *convergence relation*, here denoted $\hookrightarrow_F$, on the states of $\mathcal{P}$; this is a relation with the property that if $s \hookrightarrow_F t$ and $t \in \Diamond F$ then also $s \in \Diamond F$. From this property it follows that $Pre^*(\hookrightarrow_F, F) \subseteq \Diamond F$ for any convergence relation $\hookrightarrow_F$. The construction of $\hookrightarrow_F$ depends in general on F. Since $\hookrightarrow_F$ will be employed in a predecessor calculation, it is natural to allow the use of predecessor calculations also in the construction of $\hookrightarrow_F$ itself, but to avoid computations of transitive closures or other more powerful techniques.

Our main technique for constructing $\hookrightarrow_F$ uses a commutativity argument to infer that it satisfies the required properties. To explain its intuition, consider the following simple program, which consists of two deterministic processes executing in parallel.

$$\begin{array}{llll} \alpha_1 : & x := x - 1 & \text{if} & x > 0 \\ \alpha_2 : & y := y - 1 & \text{if} & y > 0 \end{array}$$

Variables x and y assume values in the natural numbers. For $i = 1, 2$, process i repeatedly performs action α_i. Both α_1 and α_2 are fair actions. The transition relation is the union of both actions plus the identity relation. The set F of terminated states is the single state with $x = y = 0$.

In this example, our method computes $\hookrightarrow_F$ as $\alpha_1 \cup \alpha_2$. Our method implicitly ascertains that $\hookrightarrow_F$ is a convergence relation using a commutativity argument. To understand why α_1 is in $\hookrightarrow_F$, assume that $s \xrightarrow{\alpha_1} t$ and $t \in \Diamond F$. Consider any computation from s. If it goes first to t we are done. Otherwise, it first consists of a sequence of executions of action α_2. During this sequence, α_1 remains enabled, and so must eventually (by fairness) be executed, leading to some state t'. Now observe that since α_1 and α_2 commute, t' is reachable from t. Since $t \in \Diamond F$ we infer, using the fact that $\Diamond F$ is a stable set, that $t' \in \Diamond F$ and hence that $s \in \Diamond F$. We conclude that termination is guaranteed for all states in $Pre^*(\hookrightarrow_F, F)$, which here is the set of all states.

The above method can prove termination for many programs with a regular structure. It is in general incomplete. For programs where the above method computes a too small under-approximation of $\Diamond F$, we offer the following two ways to proceed.

The backwards reachability computation can be repeated several times. If one computation produces an under-approximation G of $\Diamond F$, the next application of our method will compute $\Diamond G$ using a convergence relation $\hookrightarrow_G$ that is larger than in the first computation, since it depends on G instead of F. Let us illustrate this by changing the above program by changing the guard of α_1 into $0 < x \leq y \vee y = 0$. This destroys commutativity between α_1 and α_2 in case $y = x$. However, a first backwards reachability computation will produce the set G consisting of states with $0 \leq x \leq 1$ or with $0 \leq y < x$ as an under-approximation to $\Diamond F$. A second backwards reachability computation thereafter reveals that all states are in $\Diamond G$, hence also in $\Diamond F$.

In many cases, the under-approximation of $\Diamond F$ computed by our method is sufficiently large that other techniques (e.g., standard techniques based on ranks or transitive closure computation) become computationally feasible. For the class of parameterized systems, we have developed a powerful method, whose only nontrivial computation is predecessor calculation, which can be used after applying the commutativity-based method.

4 Proving Termination as Backward Reachability

In this section, we formalize the methods for calculating (an under-approximation of) the set $\Diamond F$ by backwards reachability analysis, presented in the previous section. We first present the general approach, and then our main technique.

Assume a program $\langle S, \longrightarrow, \mathcal{A} \rangle$. Let F be a stable set of terminated states. Define a *convergence relation* on S for F to be a relation $\hookrightarrow_F$ on S such that whenever $s \hookrightarrow_F t$ and $t \in \Diamond F$ then also $s \in \Diamond F$. The point of convergence relations is that if $\hookrightarrow_F$ is a convergence relation for F, then $\mathit{Pre}^*(\hookrightarrow_F, F) \subseteq \Diamond F$, i.e., we can use predecessor calculation to prove that termination is guaranteed from a set of states. Larger convergence relations allow to prove termination for larger sets of states. Furthermore, even if we cannot precisely calculate $\mathit{Pre}^*(\hookrightarrow_F, F)$, any under-approximation of this set is also in $\Diamond F$.

To apply these ideas, we need techniques to compute sufficiently powerful convergence relations. Any number of convergence relations can be combined into one, since the union of two convergence relations is again a convergence relation. Now we present our main technique, which is based on a commutativity argument.

Definition 1. *Let α be a deterministic fair action, and let F be a set of states. Define the* left moving states for (α, F)*, denoted $\mathit{Left}(\alpha, F)$, as the set of states s satisfying*

- *whenever there are states s', t' with $t' \notin F$ such that $s \longrightarrow s' \xrightarrow{\alpha} t'$, then there is a state t with $s \xrightarrow{\alpha} t \longrightarrow t'$.*

Intuitively, α can "move left" of $\longrightarrow$, and still reach the same state. The definition is illustrated in Figure 1.

$$\begin{tikzcd}
 & \exists\, t \arrow[r] & t' \notin F \\
\forall\, s', t' & & \\
 & s \arrow[uu, "\alpha"'] \arrow[r] & s' \arrow[uu, "\alpha"']
\end{tikzcd}$$

Fig. 1. $s \in \mathit{Left}(\alpha, F)$. Action α commutes left at state s.

Definition 2. *Define the* α-helpful states*, denoted $\mathit{Helpful}(\alpha, F)$, as the largest set T of states such that $T \subseteq ((En(\alpha) \cap \mathit{Left}(\alpha, F)) \cup F)$, and*

- *whenever $s \in \mathit{Helpful}(\alpha, F)$ and $s \longrightarrow s'$ then either $s \xrightarrow{\alpha} s'$, or $s' \in F$, or $s' \in \mathit{Helpful}(\alpha, F)$.*

Intuitively, a state is α-helpful if the properties that α is enabled and left moving remain true when any sequence of transitions not in α are taken, unless F is reached. The above concepts can be used to define a convergence relation as follows.

Theorem 1. *Let α be a fair action of $\langle S, \longrightarrow, \mathcal{A} \rangle$ and F be a stable set of states. Then the relation $\overset{\alpha}{\hookrightarrow}_F$, defined by*

$$\overset{\alpha}{\hookrightarrow}_F \;\equiv\; \alpha \wedge \mathit{Helpful}(\alpha, F)$$

is a convergence relation for F.

Proof. Assume that $s \overset{\alpha}{\hookrightarrow}_F t$ and $t \in \Diamond F$. Consider any computation $s_0\ s_1\ s_2\ \ldots$ from $s = s_0$. We must show that it contains a state in F.

- If there is a k with $s_k \in F$ we are done.
- Otherwise, if there is a k with $s_k \xrightarrow{\alpha} s_{k+1}$, let k be the least such index. By induction, using the definition of $\mathit{Helpful}(\alpha, F)$, we infer that $s_i \in \mathit{Helpful}(\alpha, F)$, hence $s_i \in En(\alpha)$ and $s_i \in \mathit{Left}(\alpha, F)$ for $i = 0, \ldots, k$. Let t_i be the unique state with $s_i \xrightarrow{\alpha} t_i$, in particular $s_{k+1} = t_k$. By induction we infer, using the definition of $\mathit{Left}(\alpha, F)$, that t_i is reachable from t for all i with $0 \leq i \leq k$. In particular, $s_{k+1} = t_k$ is reachable from t. From $t \in \Diamond F$ we infer $s_{k+1} \in \Diamond F$ and hence the computation must contain a state in F. An illustration of this argument is provided in Figure 2.
- Otherwise, we infer by induction over k, using $s \in \mathit{Helpful}(\alpha, F)$, that α is enabled in all states of the computation. By fairness, α will eventually be executed, and we are back to the previous case. $\Box$

$$\begin{tikzcd}
t \arrow[r] & t_1 \arrow[r] & t_2 \arrow[r] & \cdots \arrow[r] & t_k \\
s \arrow[u, "\alpha"'] \arrow[r, "\neg\alpha"] & s_1 \arrow[u, "\alpha"'] \arrow[r, "\neg\alpha"] & s_2 \arrow[u, "\alpha"'] \arrow[r, "\neg\alpha"] & \cdots \arrow[r, "\neg\alpha"] & s_k \arrow[u, "\alpha"']
\end{tikzcd}$$

Fig. 2. $(s, t) \in \overset{\alpha}{\hookrightarrow}_F$. The α-successor of any successor of s, is either a successor of t, or in F.

Corollary 1. $\mathit{Pre}^*(\{\overset{\alpha}{\hookrightarrow}_F | \alpha \in \mathcal{A}\}\ ,\ F) \subseteq \Diamond F$

In order to show how termination can be proven by backwards reachability analysis, we must finally explain how to compute $\mathit{Helpful}(\alpha, F)$, or an under-approximation of it, by backwards reachability analysis. We first observe that:

$$\mathit{Left}(\alpha, F) = \neg \mathit{Pre}((\longrightarrow \circ\, \alpha) - (\alpha \circ \longrightarrow), \neg F)$$

Proposition 1. *The set $\mathit{Helpful}(\alpha, F)$ is the complement of the set*

$$\mathit{Pre}^*((\mathcal{A} - \alpha) \wedge \neg F\ \ ,\ \ (\neg \mathit{Left}(\alpha, F) \cup \neg En(\alpha))\ \cap\ \neg F)$$

Proof. According to Definition 2, a state s is not in $\mathit{Helpful}(\alpha, F)$ if and only if there is a sequence of transitions from s, none of which is in α or visits a state in F, which leads to a state neither in F nor in $En(\alpha)\ \cap\ \mathit{Left}(\alpha, F)$; exactly what the proposition formalizes. $\Box$

5 Examples

In this section we illustrate our method, by applying it to two examples from the literature.

5.1 Any-Down

The example *Any-Down* is used by Podelski and Rybalchenko [31] to illustrate their method of transition invariants. In fact, our method can handle, in two iterations or less, all the examples given in [31]. For readability, we reformulate the program into the action-based syntax of the example in Section 3, as follows.

$$\begin{array}{lll} \alpha_1 : y := y+1 & \text{if} & x = 1 \\ \alpha_2 : x := 0 & \text{if} & \mathit{true} \\ \alpha_3 : y := y-1 & \text{if} & x = 0 \wedge y > 0 \end{array}$$

The program variable y assumes values in the natural numbers, and the variable x assumes values in $\{0, 1\}$. Both α_2 and α_3 are fair actions. The transition relation is the union of all three actions plus the identity relation. The set F of terminated states is the single state with $x = y = 0$. It is well-known that a standard termination proof for this program will require a ranking function whose range is larger than the natural numbers. This suggests that we need at least two iterations of our method to compute the set $\Diamond F$. We describe each iteration below.

In the first iteration we compute $\mathit{Helpful}(\alpha_i, F)$ for $i = 2, 3$ (we omit α_1, since it is not a fair action). These computations are summarized in the below table.

	$\mathit{En}(\alpha_i)$	$\mathit{Left}(\alpha_i, F)$	$\mathit{Helpful}(\alpha_i, F)$
α_2	true	$x = 0$	$x = 0$
α_3	$x = 0 \wedge y > 0$	$x = 0 \vee y = 0 \vee y = 1$	$x = 0$

We explain the entries of the table for α_2. The corresponding entries for α_3 can be explained in a similar manner. The set $\mathit{Left}(\alpha_2, F)$ includes all states s where $x = 0$. This is since either (i) $y = 0$ in which case $s \in F$; or (ii) $y > 0$, which means that α_1 is not enabled, and α_2 commutes with α_3. On the other hand, $\mathit{Left}(\alpha_2, F)$ does not include any state s with $x = 1$, as follows. We have $s \xrightarrow{\alpha_1} \xrightarrow{\alpha_2} t$, for some t with $y > 0$. Obviously, $t \notin F$ and furthermore it is not the case that $s \xrightarrow{\alpha_2} \xrightarrow{\alpha_1} t$ since α_2 disables α_1. This means we have violated the condition for being a left mover.

The set $\mathit{Helpful}(\alpha_2, F)$ includes all states where $x = 0$; such a state s belongs to $\mathit{Left}(\alpha_2, F)$. The action α_2 is enabled from s. Furthermore, the action α_1 is disabled, while the execution of α_3 from s again leads to a state satisfying $\mathit{Helpful}(\alpha_2, F)$.

By Corollary 1, the following set is in $\Diamond F$:

$$G \equiv \mathit{Pre}^*((\alpha_2 \wedge \mathit{Helpful}(\alpha_2, F)) \cup (\alpha_3 \wedge \mathit{Helpful}(\alpha_3, F)), F) \quad \equiv \quad x = 0$$

In the second iteration we compute $\mathit{Helpful}(\alpha_i, G)$ for $i = 2, 3$ in the same way. The interesting difference is that $\mathit{Left}(\alpha_2, G)$, which is true, is larger than $\mathit{Left}(\alpha_2, F)$, since any execution of α_2 leads to G. Hence also $\mathit{Helpful}(\alpha_2, G)$, which is true, is larger than $\mathit{Helpful}(\alpha_2, F)$.

	$\mathit{En}(\alpha_i)$	$\mathit{Left}(\alpha_i, G)$	$\mathit{Helpful}(\alpha_i, G)$
α_2	true	true	true
α_3	$x = 0 \wedge y > 0$	true	$x = 0$

By Corollary 1, the following set is in $\Diamond G$, hence in $\Diamond F$:

$$\mathit{Pre}^*((\alpha_2 \wedge \mathit{true}) \cup (\alpha_3 \wedge \mathit{Helpful}(\alpha_3, F)), G) \quad \equiv \quad \mathit{true}$$

5.2 Alternating Bit Protocol

We consider the *alternating bit protocol* as a second example. This protocol consists of finite-state processes that communicate over unbounded and lossy FIFO channels. As shown in our earlier work, it is decidable whether such a protocol satisfies a safety property [5], but undecidable whether a protocol satisfies a liveness property [4]. Using our technique, we can prove liveness properties for some of these protocols.

The alternating bit protocol involves a *sender* and a *receiver* that communicate over two channels c_M and c_A. Channel c_M is used to transmit messages from the *sender* to the *receiver*, and channel c_A to transmit acknowledgments from the *receiver* to the *sender*. Both channels are FIFO and faulty in the sense that messages may be lost but not reordered. The purpose of the protocol is to transmit messages from the *sender* to the *receiver* in correct order, in spite of the fact that the channels can lose messages. Corruption of messages can also be taken into account by modeling it as a loss (some mechanism will detect and discard a corrupted message). Each channel is "fair" in the sense that if infinitely many messages are input, then infinitely many messages will be delivered.

We describe the operations of *sender* and *receiver* in the protocol. At one extremity of the channels, the *sender* constructs a message m_i by adding a sequence number i in $\{0, 1\}$ to a pending message m, and sends it on the channel c_M to the *receiver*. The *sender* awaits for an acknowledgment a_i with the same sequence number on the channel c_A. If a_i arrives, the procedure is repeated with the next pending message but with an inverted sequence number $(1 - i)$. If no acknowledgment a_i arrives within a specified time period the *sender* retransmits the message m_i. Retransmissions are repeated until an acknowledgment a_i with a corresponding sequence number arrives. Acknowledgments with non-corresponding sequence numbers are discarded. On the other extremity of the channels, the *receiver* receives messages m_i from the incoming channel c_M. A message m_i is delivered if the corresponding sequence number i was expected. After delivery of m_i, the *receiver* sends on channel c_A an acknowledgment a_i with the same sequence number to the *sender*. The *receiver* expects a message with an inverted sequence number $(1 - i)$. Messages with non-expected sequence numbers are discarded.

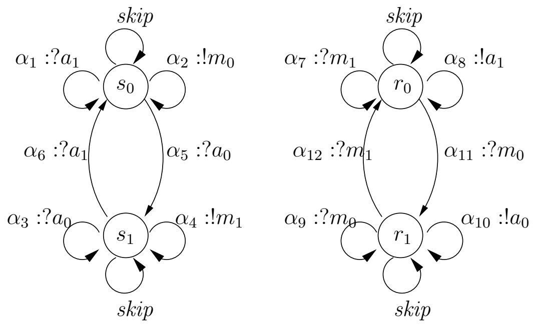

Fig. 3. The Alternating Bit Protocol

The *sender* and the *receiver* are modeled by the finite-state processes depicted in figure 3. The states of the *sender* are in $\{s_0, s_1\}$, while those of the *receiver* are in $\{r_0, r_1\}$. A state of the system is of form $s_k r_l(w_M, w_A)$ where s_k is a sender state, r_l is a receiver state, w_M is the content of channel c_M, and w_A is the content of channel c_A. The initial state is $s_0 r_0(\langle\rangle, \langle\rangle)$ with both channels empty. From a state $s_k r_l(w_M, w_A)$, the effects of the actions $!m_i$ and $!a_i$, with i in $\{0, 1\}$, are respectively $s_k r_l(m_i \bullet w_M, w_A)$ and $s_k r_l(w_M, a_i \bullet w_A)$ (the operator $\bullet$ being the concatenation of channels content). The state $s_k r_l(w_M, w_A)$ results from applying the action $?m_i$ to the state $s_k r_l(w_M \bullet m_i, w_A)$, or from applying the action $?a_i$ to the state $s_k r_l(w_M, w_A \bullet a_i)$. Here, c_M and c_A are perfect FIFO buffers, and message losses are modeled as a non-deterministic choice between a send and a *skip* action.

There are techniques to automatically calculate the set of states reachable from the initial state $s_0 r_0(\langle\rangle, \langle\rangle)$. An example is to start from the initial state and to apply the technique of *loop-first search* [**?**]. This technique generates the set of reachable states by taking all possible transitions, and evaluating (whenever possible) the effect of performing an arbitrary number of the same transition. Examples of such *accelerations* are α_1^* or α_2^*, resulting in the addition of to the tail of c_M, respectively the substraction from the head of c_A, of an arbitrary number of m_0, respectively a_1. Sets of states can be captured by *Queue-content Decision Diagram (QDD)* of the form $s_k r_l(w_M, w_A)$ where w_M and w_A are regular languages. The search stops once the set of generated states stabilizes, i.e. no new states are generated when applying transitions. For the protocol at hand, this technique returns the set of reachable states as union of the four sets $s_0 r_0(m_0^* m_1^*, a_1^*)$, $s_0 r_1(m_0^*, a_0^* a_1^*)$, $s_1 r_0(m_1^*, a_1^* a_0^*)$, and $s_1 r_1(m_1^* m_0^*, a_0^*)$.

We describe the program $\langle S, \longrightarrow, \mathcal{A}\rangle$ corresponding to the *alternating bit protocol*. The set S is here chosen to be the set of reachable configurations computed above. The transition relation $\longrightarrow$ is the union of the actions *skip* and $\alpha_1, \ldots, \alpha_{12}$. All these actions, except *skip*, are in $\mathcal{A}$. This corresponds to the assumption that if a message is continuously retransmitted, then eventually one of the messages is not lost.

We use the method defined in Section 4 to prove the following four progress properties of the protocol.

$$\begin{array}{ll}
P_{r_0 r_1}: & s_0 r_0(m_0^* m_1^*, a_1^*) \subseteq \Diamond s_0 r_1(m_0^*, a_0^* a_1^*) \\
P_{s_0 s_1}: & s_0 r_1(m_0^*, a_0^* a_1^*) \subseteq \Diamond s_1 r_1(m_1^* m_0^*, a_0^*) \\
P_{r_1 r_0}: & s_1 r_1(m_1^* m_0^*, a_0^*) \subseteq \Diamond s_1 r_0(m_1^*, a_1^* a_0^*) \\
P_{s_1 s_0}: & s_1 r_0(m_1^*, a_1^* a_0^*) \subseteq \Diamond s_0 r_0(m_0^* m_1^*, a_1^*)
\end{array}$$

Observe the property $P_{r_0 r_1}$ implies that from any state in the set $s_0 r_0(m_0^* m_1^*, a_1^*)$, the system is guaranteed to reach a state in $s_0 r_1(m_0^*, a_0^* a_1^*)$. This means the *receiver* changed state from r_0 to r_1. In other words, the receiver is guaranteed to take action α_{11} and to receive the message m_0. A similar reasoning with $P_{s_0 s_1}$, $P_{r_1 r_0}$ and $P_{s_1 s_0}$ ensures *sender* and *receiver* indefinitely alternate sending m_0, a_0, m_1 and a_1. We show in the following how to prove the property $P_{r_0 r_1}$; the other progress properties are proven in a similar manner.

Assume the set F coincides with $s_0 r_1(m_0^*, a_0^* a_1^*)$. We use the method defined in Section 4 to calculate a set of states included in $\Diamond F$. To ensure the stability of F, we first modify all actions α to $\alpha \wedge \neg F$. Observe $\alpha_5 \wedge \neg F$ is an empty relation, while the other actions are modified as shown in figure **??**. The results of the computations (according to Proposition 1) of the helpful set of states for each fair action α_i in $\mathcal{A}$ appear in the same figure. Let us give an intuition of why $\mathit{Helpful}(\alpha_7, F)$ coincides with the union of the sets $s_0r_0(m_0^*m_1^+, a_1^*)$, $s_1r_0(m_1^+, a_1^*a_0^*)$ and $s_0r_1(m_0^*, a_0^*a_1^*)$. For every state s in this union, it is the case that either (i) s is in $F = s_0r_1(m_0^*, a_0^*a_1^*)$; or (ii) s is in the union of $s_0r_0(m_0^*m_1^+, a_1^*)$ and $s_1r_0(m_1^+, a_1^*a_0^*)$. In the second case, observe that α_7 is enabled and that the only action that does not commute with α_7 is action α_{11} (which is not enabled). We have that (i) α_7 is enabled from s and commutes with any other enabled action; and (ii) the execution of any other action from s leads to the same union of $s_0r_0(m_0^*m_1^+, a_1^*)$ and $s_1r_0(m_1^+, a_1^*a_0^*)$. Observe that α_7 is not enabled outside the union of $s_0r_0(m_0^*m_1^+, a_1^*)$, $s_1r_0(m_1^+, a_1^*a_0^*)$ and $s_0r_1(m_0^*, a_0^*a_1^*)$. Therefore, this union coincides with $\mathit{Helpful}(\alpha_7, F)$. The other sets in figure **??** can be explained in a similar manner.

	$\mathit{En}(\alpha_i)$	$\mathit{En}(\alpha_i) \cap \mathit{Left}(\alpha_i, F)$	$\mathit{Helpful}(\alpha_i, F)$
α_1	$s_0r_0(m_0^*m_1^*, a_1^+)$	$s_0r_0(m_0^*m_1^*, a_1^+)$	$s_0r_0(m_0^*m_1^*, a_1^+) \cup s_0r_1(m_0^*, a_0^*a_1^*)$
α_2	$s_0r_0(m_0^*m_1^*, a_1^*)$	$s_0r_0(m_0^*m_1^*, a_1^*)$	$s_0r_0(m_0^*m_1^*, a_1^*) \cup s_0r_1(m_0^*, a_0^*a_1^*)$
α_3	$s_1r_0(m_1^*, a_1^*a_0^+)$ $\cup\, s_1r_1(m_1^*m_0^*, a_0^+)$	$s_1r_0(m_1^*, a_1^*a_0^+)$ $\cup\, s_1r_1(m_1^*m_0^*, a_0^+)$	$s_1r_0(m_1^*, a_1^*a_0^+) \cup s_1r_1(m_1^*m_0^*, a_0^+)$ $\cup\, s_0r_1(m_0^*, a_0^*a_1^*)$
α_4	$s_1r_0(m_1^*, a_1^*a_0^*)$ $\cup\, s_1r_1(m_1^*m_0^*, a_0^*)$	$s_1r_0(m_1^*, a_1^*a_0^*)$ $\cup\, s_1r_1(m_1^*m_0^*, a_0^*)$	$s_1r_1(m_1^*m_0^*, a_0^*) \cup s_0r_1(m_0^*, a_0^*a_1^*)$
α_5	$\emptyset$	$\emptyset$	$s_0r_1(m_0^*, a_0^*a_1^*)$
α_6	$s_1r_0(m_1^*, a_1^+)$	$\emptyset$	$s_0r_1(m_0^*, a_0^*a_1^*)$
α_7	$s_0r_0(m_0^*m_1^+, a_1^*)$ $\cup\, s_1r_0(m_1^+, a_1^*a_0^*)$	$s_0r_0(m_0^*m_1^+, a_1^*)$ $\cup\, s_1r_0(m_1^+, a_1^*a_0^*)$	$s_0r_0(m_0^*m_1^+, a_1^*) \cup s_1r_0(m_1^+, a_1^*a_0^*)$ $\cup\, s_0r_1(m_0^*, a_0^*a_1^*)$
α_8	$s_0r_0(m_0^*m_1^*, a_1^*)$ $\cup\, s_1r_0(m_1^*, a_1^*a_0^*)$	$s_0r_0(m_0^*m_1^*, a_1^*)$ $\cup\, s_1r_0(m_1^*, a_1^*a_0^*)$	$s_0r_0(m_0^*m_1^*, a_1^*) \cup s_1r_0(m_1^*, a_1^*a_0^*)$ $\cup\, s_0r_1(m_0^*, a_0^*a_1^*)$
α_9	$s_1r_1(m_1^*m_0^+, a_0^*)$	$s_1r_1(m_1^*m_0^+, a_0^*)$	$s_1r_1(m_1^*m_0^+, a_0^*) \cup s_0r_1(m_0^*, a_0^*a_1^*)$
α_{10}	$s_1r_1(m_1^*m_0^*, a_0^*)$	$s_1r_1(m_1^*m_0^*, a_0^*)$	$s_0r_1(m_0^*, a_0^*a_1^*)$
α_{11}	$s_0r_0(m_0^+, a_1^*)$	$s_0r_0(m_0^+, a_1^*)$	$s_0r_0(m_0^+, a_1^*) \cup s_0r_1(m_0^*, a_0^*a_1^*)$
α_{12}	$s_1r_1(m_1^+, a_0^*)$	$\emptyset$	$s_0r_1(m_0^*, a_0^*a_1^*)$

Fig. 4. Calculation of $\Diamond(s_0r_1(m_0^*, a_0^*a_1^*))$

By Corollary 1, the set $G = \mathit{Pre}^*(\{\overset{\alpha_i}{\hookrightarrow}_F | i = 1, \ldots, 12\},\ F)$ is in $\Diamond F$. Observe that $s_0r_0(m_0^*m_1^*, a_1^*) = \mathit{Pre}^*(\{\alpha_i \wedge \mathit{Helpful}(\alpha_i, F) | i = 2, 7, 11\}, s_0r_0(m_0^*, a_1^*))$ is a subset of G. We therefore conclude that $s_0r_0(m_0^*m_1^*, a_1^*) \quad \subseteq \quad \Diamond F$.

6 Parameterized Systems

In this section we consider verification of liveness properties for parameterized systems: these are systems with an arbitrary number of similar processes operating in parallel. A challenge is that they are not finite-state, since the number of processes is unbounded.

We describe an implementation of our method in the framework of Regular Model Checking [7]. For several examples, the proof rule of Section 4 computes a strict under-approximation of the set $\Diamond F$; therefore we also present a complementary rule which can prove termination for those examples.

Example: Szymanski's Algorithm As an example of a parameterized system, we describe the mutual exclusion algorithm by Szymanski [33]. In the algorithm, an arbitrary number of processes compete for a critical section. The processes are numbered, say from 1 to N. The *local state* of each process consists of a control state ranging over the integers from 1 to 7 and of two Boolean flags, w and s. A pseudo-code description of the behavior of process number i is shown in Figure 4.

```
1:   await $\forall j : j \neq i : \neg s[j]$
2:   $w[i], s[i] := true, true$
3:   if $\exists j : j \neq i : (pc[j] \neq 1) \wedge \neg w[j]$
         then $s[i] := false$ ; goto 4
         else $w[i] := false$ ; goto 5
4:   await $\exists j : j \neq i : s[j] \wedge \neg w[j]$
         then $w[i], s[i] := false, true$
5:   await $\forall j : j \neq i : \neg w[j]$
6:   await $\forall j : j < i : \neg s[j]$
7:   $s[i] := false$ ; goto 1
```

Fig. 5. Szymanski

For instance, according to the code on line 6, if the control state of a process i is 6, and if the value of s is *false* for all processes $j < i$, then the control state of i may be changed to 7. Line 7 represents the critical section. Each numbered line is modeled as an action: $\alpha_j(i)$ is the statement beginning at line j in the pseudo-code for process i. All actions are fair, except $\alpha_1(i)$; this action represents process i entering the competition for the critical section, and therefore its execution should not be enforced.

Starvation freedom can be formulated as follows: whenever any process is at line 2 it will eventually reach line 7. Define F_k to be all states in which process k is at line 7. To prove starvation freedom for process k we must show that all reachable states where process k is at line 2 are in $\Diamond F_k$.

6.1 A Complementary Termination Rule

In this section, we present a proof rule for termination, which is particularly suitable for the class of parameterized systems considered in this section. It will be used to complement the method of Corollary 1. The rule assumes that we select a finite number of fair actions of the program, and establishes that a state s is in $\Diamond F$ if computations from s satisfy

– whenever one of these actions is enabled, it remains enabled until it is executed,

– each of the actions can be executed at most once before F is reached, and
– when all these actions are disabled, the computation has reached F.

This rule is particularly useful for parameterized systems, since termination is often achieved by letting a selected subset of the processes execute a fixed sequence of actions (i.e., statements). Let us define the involved properties formally. Assume a program $\langle S, \longrightarrow, \mathcal{A} \rangle$. Let F be a set of terminated states.

– $\mathit{Persist}(\alpha, F)$ is the set of states s such that in any computation from s, whenever α is enabled, it remains enabled unless it is executed or F is reached.
– $\mathit{Twice}(\alpha, F)$ is the set of states, from which there exists a computation where α is executed twice (or more) without visiting F.
– Let $\mathcal{B}$ be a finite set of actions. $\mathit{After}(\mathcal{B}, F)$ is the set of states s such that in any computation from s, whenever all actions in $\mathcal{B}$ are disabled at a state s', then $s' \in F$.

The above sets are computable using backwards reachability analysis, in a manner analogous to the way $\mathit{Helpful}(\alpha, F)$ is computed in Proposition 1. Note that the set $\mathcal{B}$ used in $\mathit{After}(\mathcal{B}, F)$ is typically a parameterized set of actions, containing a set of actions of form $\alpha_j(i)$ for a finite set j, and an arbitrary i with $1 \leq i \leq N$. Thus the set $\mathcal{B}$ is unboundedly large, but still finite. Care must be taken to handle the parameters correctly when performing the predecessor calculations. Now we state the termination rule.

Theorem 2. *Let $\mathcal{B}$ be a set of fair actions of $\langle S, \longrightarrow, \mathcal{A} \rangle$, and let F be a set of states in S. Then*

$$\left[\mathit{After}(\mathcal{B}, F) \;\cap\; \bigcap_{\alpha \in \mathcal{B}} (\neg \mathit{Twice}(\alpha, F) \;\cap\; \mathit{Persist}(\alpha, F)) \right] \subseteq \Diamond F$$

Proof. Let s be a state in the set defined by the left-hand side. Consider a computation from s. Assume that it contains no state in F. Then, since $s \in \mathit{After}(\mathcal{B}, F)$ it also contains no state in which all actions in $\mathcal{B}$ are disabled. This means that at any state in the computation, some action α is enabled. Since $s \in \mathit{Persist}(\alpha, F)$ the action α will remain enabled until it is executed, and thereafter (since $s \in \neg \mathit{Twice}(\alpha, F)$) never be executed again. This implies that after a finite number of computation steps, all actions in $\mathcal{B}$ have been executed. This contradicts the previous conclusion that thereafter some action in $\mathcal{B}$ is enabled, and will eventually be executed.

6.2 Implementation

We have implemented a verification method based on Corollary 1 and Theorem 2 in the framework of Regular Model Checking [7], and applied it to a number of well-known parameterized mutual exclusion protocols.

Verification Procedure For each protocol, we have modeled F_k as the set of states where process k is in the critical section. We have thereafter computed an under-approximation G_k of $\Diamond F_k$ using the method of Section 4, and thereafter applied the complementary rule described in Section 6.1 to compute $\Diamond G_k$. To ensure that predecessors are reachable states, we computed the set of (forwards) reachable states, and restricted the actions to it.

In our experiments we manually chose what rules to apply and when, to test their expressive power. However, the approach may be fully automated by e.g. applying the rules alternatingly. As a termination condition one could use that the complementary rule does not increase the set $\Diamond F$, no matter which action it is applied to.

As an example, we describe how our verification of starvation freedom for Szymanski's algorithm works. Three successive applications of Corollary 1 establish starvation freedom for almost all the system states where process k is waiting. However, Corollary 1 cannot prove starvation freedom for system states where there are processes at both line 1 and line 2. The reason for this is that the actions of line 2 may disable the actions on line 1, thereby destroying commutativity. By using also one application of Theorem 2, starvation freedom is proven for all the system states where process k is waiting, as desired.

Results The verification results of our implementation are presented in Table 1. We have computed the sets of states from which starvation freedom for process k is guaranteed, as a set which depends on k. In all cases, the computed *live states* contain all the terminating states. For example, the live states of Szymanski's algorithm are: "whenever process k is at line 2". The column "Time" contains time measured from our implementation. The experiments were run on a PC with a 2.4 GHz processor and 1 GB of RAM. For the first three protocols, we need apply only Corollary 1. For the last three last protocols, we need also Theorem 2. Dijkstra's algorithm takes significantly longer time to verify because it contains an action where a global variable is set. Computing the effect of arbitrarily many executions of such an action is relatively expensive in our current implementation [7].

Model	Token Pass	Token Ring	Bakery	Szymanski	Burns	Dijkstra
Time	9 s	14 s	36 s	7 min 15 s	7 min 30 s	55 min 11 s

Table 1. Experimental results.

Comparison with Related Work Several works have considered verification of individual starvation freedom for parameterized mutual exclusion protocols. In papers [30, 10] the Szymanski protocol and the Bakery protocol are verified in 95.87 seconds and 9 seconds respectively, using manually supplied abstractions. The works [21, 22] verify the Bakery protocol using automatically generated ranking functions, but do not report running times. We have previously verified the Bakery protocol in 44.2 seconds using repeated reachability [27], on the same system. To our knowledge, starvation freedom for the algorithms of Burns and Dijkstra has not been successfully automatically verified before.

Techniques exist for quicker accelerations, which should significantly improve the performance ([1, 29]). There is a need for quick automatic accelerations, which also cover global variables and compositions of actions.

7 Conclusions

We have presented a method for proving liveness and termination properties of fair concurrent programs using backwards reachability analysis. The method uses neither computation of transitive closure nor explicit construction of ranking functions and helpful directions, and relies instead on showing certain commutativity properties between different actions of the program. The advantage of our method is that reachability analysis can typically be expected to be simpler to perform than computation of transitive closures or ranking functions. We expect that it should be possible to use and develop powerful techniques for backwards reachability analysis for many classes of parameterized and infinite-state programs. The technique is in general incomplete, but its power can be increased by performing repeated applications and by applying complementary techniques afterwards. The examples in the paper indicate that the method should be applicable to several classes of infinite-state systems. In particular, we have shown that our technique is able to prove starvation-freedom for several parameterized mutual exclusion protocols, for which automated techniques have previously been too expensive.

References

1. P. A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling global conditions in parameterized system verification. In *Proc. 11^{th} Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 134–145, 1999.
2. P. A. Abdulla, K. Čerāns, B. Jonsson, and T. Yih-Kuen. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160:109–127, 2000.
3. P. A. Abdulla, A. Collomb-Annichini, A. Bouajjani, and B. Jonsson. Using forward reachability analysis for verification of lossy channel systems. *Formal Methods in System Design*, 25(1):39–65, 2004.
4. P. A. Abdulla and B. Jonsson. Undecidable verification problems for programs with unreliable channels. *Information and Computation*, 130(1):71–90, 1996.
5. P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996.
6. P. A. Abdulla, B. Jonsson, M. Nilsson, J. d'Orso, and M. Saksena. Regular model checking for LTL(MSO). In *Proc. 16^{th} Int. Conf. on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, 2004.
7. P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A survey of regular model checking. In *Proc. CONCUR 2004, 14^{th} Int. Conf. on Concurrency Theory*, volume 3170 of *LNCS*, pages 35–48, 2004.
8. I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In R. Cousot, editor, *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, pages 164–180. Springer Verlag, 2005.
9. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 2001*, pages 203–213, 2001.

10. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proc.* 16^{th} *Int. Conf. on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, 2004.
11. A. Bradley, Z. Manna, and H. Sipma. Linear ranking with reachability. In M. Abadi and L. de Alfaro, editors, *Proc. CONCUR 2005,* 15^{th} *Int. Conf. on Concurrency Theory*, volume 3653 of *Lecture Notes in Computer Science*, pages 491–504, 2005.
12. A. Bradley, Z. Manna, and H. Sipma. Termination analysis of integer linear loops. In K. Etessami and S. Rajamani, editors, *Proc.* 17^{th} *Int. Conf. on Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 488–502, 2005.
13. A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In R. Cousot, editor, *Proc. VMCAI 2005, Verification, Model Checking, and Abstract Interpretation, 6th International Conference, Paris, January 17-19*, volume 3385 of *Lecture Notes in Computer Science*, pages 113–129. Springer Verlag, 2005.
14. K. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
15. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
16. E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *Software Tools for Technology Transfer*, 2:279–287, 1999.
17. M. Colon and H. Sipma. Synthesis of linear ranking functions. In T. Margaria and W. Yi, editors, *Proc. TACAS '01,* 7^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 67–81. Springer Verlag, 2001.
18. M. Colon and H. Sipma. Practical methods for proving program termination. In Brinskma and Larsen, editors, *Proc.* 14^{th} *Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 442–454. Springer Verlag, 2002.
19. B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In C. Hankin and I. Siveroni, editors, *Proc.* 12^{th} *Int. Symp. on Static Analysis*, volume 3672 of *LNCS*, pages 87–101. Springer Verlag, 2005.
20. P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In R. Cousot, editor, *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, pages 1–24. Springer Verlag, 2005.
21. Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with incomprehensible ranking. In K. Jensen and A. Podelski, editors, *Proc. TACAS '04,* 10^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 482–496. Springer Verlag, 2004.
22. Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with invisible ranking. In B. Steffen and G. Leiv, editors, *Proc. VMCAI 2004, Verification, Model Checking, and Abstract Interpretation, 5th International Conference, Venice, January 11-13*, volume 2937 of *Lecture Notes in Computer Science*, pages 223–238. Springer Verlag, 2004.
23. G. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.
24. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proc.* 28^{th} *ACM Symp. on Principles of Programming Languages*, pages 81–92, 2001.
25. Z. Manna and A. Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming*, 4(4):257–289, 1984.
26. Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. In R. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative*, pages 125–159. ACM Press and Addison-Wesley, 1991.
27. M. Nilsson. *Regular Model Checking*. PhD thesis, Uppsala University, 2005.

28. A. Pnueli, A. Podelski, and A. Rybalchenko. Separating fairness and well-foundedness for the analysis of fair discrete systems. In N. Halbwachs and L. Zuck, editors, *Proc. TACAS '05,* 11^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 124–139. Springer Verlag, 2005.
29. A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 328–343. Springer Verlag, 2000.
30. A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,infinity)-counter abstraction. In *Proc.* 14^{th} *Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, 2002.
31. A. Podelski and A. Rybalchenko. Transition invariants. In *Proc. LICS' 04* 20^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 32–41, 2004.
32. A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *Proc.* 32^{th} *ACM Symp. on Principles of Programming Languages*, pages 132–144, 2005.
33. B. K. Szymanski. Mutual exclusion revisited. In *Proc. Fifth Jerusalem Conference on Information Technology*, pages 110–117, Los Alamitos, CA, 1990. IEEE Computer Society Press.
34. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. LICS '86,* 1^{st} *IEEE Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.
35. P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97, Vancouver, July 1998. Springer Verlag.