Using Fairness to Make Abstractions Work

Dragan Bošnački[1], Natalia Ioustinova[2], and Natalia Sidorova[1]

[1] Eindhoven University of Technology
Den Dolech 2, P.O. Box 513, 5612 MB Eindhoven, The Netherlands
d.bosnacki@tue.nl, n.sidorova@tue.nl
[2] Department of Software Engineering, CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Natalia.Ioustinova@cwi.nl

Abstract. Abstractions often introduce infinite traces which have no corresponding traces at the concrete level and may lead to failure of the verification. Refinement does not always help to eliminate those traces. In this paper, we consider a timer abstraction that introduces a cyclic behaviour on abstract timers and we show how one can exclude cycles by imposing a strong fairness constraint on the abstract model. By employing the fact that the loop on the abstract timer is a self-loop, we render the strong fairness constraint into a weak fairness constraint and embed it into the verification algorithm. We implemented the algorithm in the DTSpin model checker and showed its efficiency on case studies. The same approach can be used for other data abstractions that introduce self-loops.

1 Introduction

Abstraction techniques are widely used to make the verification of complex/parameterised/infinite systems feasible. Abstraction, intuitively, means replacing one semantical model by an abstract, in general, simpler one. The abstraction needs to be *safe*, which means that every property checked to be true on the abstract model, holds for the concrete one as well. This allows the transfer of positive verification results from the abstract model to the concrete one.

The concept of safe abstraction is well-developed within the *Abstract Interpretation* framework [8, 9, 12]. The relation between the concrete model and its safe abstraction is formalized there as a requirement on the relation between the data operations of the concrete system and their abstract counterparts. Every value of the concrete state space is mapped by the *abstraction function* α into an abstract value that "describes" the concrete value. As an example consider the abstraction of integers into their signs in which e.g. -3 is mapped by α into **neg**. For every operation (function) f on the concrete level, an abstraction f^α needs to be defined which "mimics" f. In general, the abstraction can be *nondeterministic*. For example, addition (+) over the integers is abstracted into an operation ($+^\alpha$) such that **pos** $+^\alpha$ **neg** may yield **pos** or **neg** nondeterministically. This is formally captured by letting f^α be a function into the powerset over the domain of abstract values.

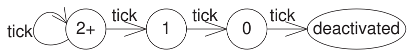

Fig. 1. Abstracted timer

Working within the Abstract Interpretation framework guarantees the preservation (in the direction from the abstract to the concrete model) of the truth of formulas of temporal logics without existential quantification over paths, e.g. $\Box L_\mu^+$ (i.e., all formulas of the μ-calculus without negation and containing only the $\Box$ operator) or next-free LTL [20, 11]. Counterexamples can be spurious. In case a counterexample is found, the abstraction should be refined and the refined model is then model-checked. Such a sequence of refinements can happen to be infinite; in this case one needs different techniques to prove or disprove the property.

In this paper we consider a simple abstraction for (discrete) timers similar to the one from [10]. This abstraction is often used to prove that a property holds for all instantiations of settings of a timer that are greater than or equal to some value k. It leaves all values below k unchanged and maps all other values to the abstract value k^+. Being a deterministic operation on the concrete model, the time progress operation *tick* becomes non-deterministic on the abstract one (see Fig. 1). That introduces infinite traces with $k^+ \xrightarrow{tick} k^+$ being chosen whenever *tick* is enabled. As a result, the timer never expires, which, in general, does not correspond to any trace of the concrete model. For instance, properties of the form $\Box(\phi \rightarrow \Diamond\psi)$ get disproved on the abstract model whenever they depend on the fact that the timer in question eventually expires after being set. Refining the model by taking a greater value for k, we still keep the loop at k_{new}^+. So, refinement gives no solution to this problem.

The systems we consider are specified as parallel compositions of communicating processes. A process consists of a number of locations, variables and a number of transitions connecting the locations and changing the valuations of variables. Processes can communicate by rendezvous/buffered message passing and through shared memory. There are explicit timing constraints in the specification imposed by timer operations.

We assume that the properties are given in the universal fragment of the μ-calculus, $\Box L_\mu$, consisting of formulas in which the negations are applied only to atomic propositions. The verification methodology we propose works for any formula of the universal fragment without negation $\Box L_\mu^+$ and, under certain conditions that occur relatively often in practice (for instance, if the formula does not refer to abstracted variables (timers)), for the whole $\Box L_\mu$.[3]

[3] Since any $\Box L_\mu$ formula can be rewritten into an equivalent $\Box L_\mu^+$ formula, and any formula of the universal unrestricted μ-calculus has an equivalent formula in $\Box L_\mu$ (see e.g. [20]), this is not a significant loss of generality.

To exclude the infinite loop $k^+ \xrightarrow{tick} k^+$ that causes spurious counterexamples, we impose a strong fairness condition Φ^α on the abstract model, which we call *t-fairness*: "For any trace where $k^+ \xrightarrow{tick} (k-1)$ is infinitely often enabled, $k^+ \xrightarrow{tick} (k-1)$ is infinitely often taken or t is infinitely often set to a new value". We show that the concrete property Φ that corresponds to the t-fairness condition Φ^α trivially holds on the concrete model. Therefore, in order to prove a formula ϕ on the concrete system we check the validity of the formula $\Phi^\alpha \rightarrow \phi^\alpha$ on the abstract one, where ϕ^α is the corresponding abstract version of ϕ. If $\Phi^\alpha \rightarrow \phi^\alpha$ holds, we conclude that ϕ holds on the concrete system. It should be emphasized though that we use fairness only to eliminate unwanted traces in the abstract system. We do not lift fairness constraints from the concrete system to the abstract system.

By exploiting some specifics of the class of systems we are working with, we show that the strong fairness criterion can be reformulated into a weak fairness one. When one deals with explicit model checking, this is often a significant advantage because algorithmically, it could be easier to deal with the latter. Moreover, when one stays in the realm of explicit-state model checking, it is much more efficient to build the t-fairness check into the model checking algorithm, instead of expressing it as a formula. In this case, one can check for the validity of ϕ on the abstract model, assuming a built-in t-fairness check. The t-fairness check algorithm we propose here is inspired by Choueka's flag algorithm [5], and it is a version of the algorithm for weak process fairness which is implemented in SPIN.

We implemented our algorithm in DTSPIN [3] (a discrete-time version of the Spin model checker [15]) and tested the prototype implementation on some examples from the literature with encouraging results.

Related work. Counter abstractions similar to the timer abstraction we use are quite standard and they can be traced to [21]. Such abstractions are often applied to abstract (discrete) timers for the verification of *safety* properties (see e.g. [10]). We study here the verification of *liveness* properties, which gives rise to the use of fairness requirements on the abstract model.

There are several papers that deal with the problem of eliminating spurious execution sequences caused by abstraction. The closest to our approach is the theory of linear abstraction from [17] (also described in [18]). The general method of data abstraction presented there can also suffer from the problem of spurious execution sequences. To eliminate those, it is suggested to augment the system under consideration by an auxiliary monitoring module (executed synchronously with the system) and then to abstract the system obtained by such a composition. In one of the examples, [17] features a three-valued counter abstraction ($\{0, 1, 2^+\}$, using our notation). Thus, one could apply the idea of a monitoring process to eliminate extra sequences introduced by self-loops to abstract states. However, this would lead to a solution based on strong fairness on the transition level. The monitor labels the "critical" transitions with -1 or $+1$. The (strong) fairness criterion requires that if a -1 transition is executed infinitely often then also a +1 transition is executed infinitely often. This ensures leaving the artificial self-loops in the abstract state space introduced by the abstraction. As it was already emphasized, we show that in the context of timer abstraction, such a straightforward strong fairness can be transformed into a weak one, which is a significant advantage in the context of explicit model checking.

In [22] the authors present a three-valued counter abstraction in the context of the verification of parameterized systems, i.e., networks of N identical concurrent processes, where N is an arbitrary finite number. The counters count the number of processes at a particular control (program) location. The solution to the problem of spurious execution sequences also in this case boils down to strong fairness. To this end two new variables *from* and *to* are introduced. The unwanted self-looping sequences are eliminated by the natural requirement that for each process location l if the processes enter l infinitely many times, then they must also leave it infinitely many times.

The problem of parameterized networks of processes is also treated in [1], with a solution for the spurious sequences which resembles both of the above given approaches. The role of the monitors from [17] is played by "ranking functions", similar to the ones used to ensure the termination of sequential programs. The ranking functions count how many processes have executed a particular transition in the concrete system. By abstracting a ranking function value, similarly to [22], one obtains a separation of the "critical" transitions into "negative" and "positive" ones. The "marking algorithm" which solves the problem of spurious sequences is based on strong fairness. The efficiency remarks in favor of our solution in the context of explicit model checking would also apply to [1] and [22].

α-Spin [13] is an extension of Spin with abstraction. The abstraction framework of α-Spin is based on the Abstract Interpretation theory and in that regard it is similar to our approach. However, to the best of our knowledge, there is no work that deals with spurious executions in the context of α-Spin. Another approach to use abstractions in combination with Spin can be found in [15].

The paper is organised as follows: In Section 2 we describe the timer abstraction and introduce the notion of t-fairness. In Section 3 we present the verification algorithm. In Section 4 we describe our implementation of t-fairness in DTSpin. In Section 5 we discuss some experimental results. Finally in Section 6 we give some conclusions.

2 Timer Abstraction and Fairness

Currently, model checkers provide some facilities to (automatically) reduce a state space, like partial-order reduction techniques. These techniques deal mainly with the control flow of a model. On the contrary, data (values stored and transmitted in a system), whose domain is often infinite or very large, are not handled by them; it is a task of a user to present data in a verification model in a finite form of reasonable size. Depending on the property to be verified, the actual values of data may sometimes be ignored or replaced by some abstract values. In an abstract model, the operations on data are mimicked by new ones on the abstract data. The main requirement for an abstraction is that the abstract system behaviour should correctly reflect the behaviour of the original system with respect to a verification task in the sense that (1) an abstraction should capture all essential points in the system behaviour, i.e., be not "too abstract", and (2) an abstraction should be safe.

Abstraction and Abstract Interpretation Framework. We first give a brief overview of the formal aspects of abstraction (see [1, 20] for more detail).

Let the semantics of a concrete system M be given with the corresponding transition system $T = (S, R)$, where S is a set of states and $R \subseteq S \times S$ is a transition relation. Given an abstract state space ${}_\alpha S$ and a total *description* relation $\rho_s \subseteq S \times {}_\alpha S$, we derive the pair of (monotonic) functions α and $\tilde{\gamma}$, where $\alpha : 2^S \rightarrow 2^{{}_\alpha S}$ is the point-wise lifting of ρ_s to the sets of states, i.e. $\alpha = post[\rho_s]$, and $\tilde{\gamma} : 2^{{}_\alpha S} \rightarrow 2^S$ is the inverse image of ρ_s, i.e. $\tilde{\gamma} = pre[\rho_s]$ (*post* and *pre* are the standard relation post- and preimage.) Intuitively, ρ_s induces a simulation relation between T and ${}_\alpha T$ (c.f. [20]). We say that ${}_\alpha T = ({}_\alpha S, {}_\alpha R)$ is an abstraction of T with regard to α, denoted as $T \sqsubseteq_\alpha {}_\alpha T$ iff $\forall Q \subseteq {}_\alpha S$: $post[R](\tilde{\gamma}(Q)) \subseteq \tilde{\gamma}(post[{}_\alpha R](Q))$. As a consequence, a given formula ϕ holds on T if it holds on ${}_\alpha T$, under condition that $\tilde{\gamma}(\mathcal{I}^\alpha(\phi)) \subseteq \mathcal{I}(\phi)$ for corresponding interpretations $\mathcal{I}, \mathcal{I}^\alpha$. The preservation result holds for formulas of temporal logics without existential quantification over paths, e.g. $\Box L_\mu^+$ or LTL [20, 11].

Given an abstraction function α, let ϕ, ϕ^α be μ-calculus formulas with the corresponding sets of atomic propositions $\mathcal{P}, \mathcal{P}^\alpha$. The semantics of ϕ and ϕ^α is given with the interpretation functions $\mathcal{I} : \mathcal{P} \rightarrow 2^S$ and $\mathcal{I}^\alpha : \mathcal{P}^\alpha \rightarrow 2^{{}_\alpha S}$, respectively. Let p^α be the proposition that corresponds to the subset $\mathcal{I}^\alpha(p) = \alpha(\mathcal{I}(p)) - \alpha(\overline{\mathcal{I}(p)})$ of the abstract state space ${}_\alpha S$. We say then that p^α is a *contracting* abstraction of p under α [17]. (Note that p^α can be considered as interpretation of p under $\mathcal{I}^\alpha : \mathcal{P} \rightarrow 2^{{}_\alpha S}$.) We call ϕ^α a contracting abstraction of formula ϕ if ϕ^α is obtained by replacing each atomic proposition p in ϕ with its contracting abstraction p^α.

Abstraction function α is *consistent*[4] with $\mathcal{I}$ iff for each $p \in \mathcal{P} : \alpha(\mathcal{I}(p)) \cap \alpha(\overline{\mathcal{I}(p)}) = \emptyset$, i.e. the images by α of the interpretations of p and $\neg p$ are not contradictory [20]. (Consistency of $\tilde{\gamma}$ with $\mathcal{I}^\alpha$ is defined analogously.) In this case, we call the contracting abstraction ϕ^α a consistent abstraction of ϕ. Note that for all $s \in S, s^\alpha \in {}_\alpha S$ such that $s^\alpha \in \alpha(\{s\})$, $s \models p$ iff $s^\alpha \models p^\alpha$ precisely when p^α is consistent, and $s \models p$ if $s^\alpha \models p^\alpha$ when p^α is contracting.

Theorem 1. *Let $T = (S, R)$ and ${}_\alpha T = ({}_\alpha S, {}_\alpha R)$ be two transition systems such that $T \sqsubseteq_\alpha {}_\alpha T$, with interpretation functions $\mathcal{I}, \mathcal{I}^\alpha$ defined as above. Given a $\Box L_\mu^+$ (resp. $\Box L_\mu$) formula ϕ, let ϕ^α be a contracting (resp. consistent with $\mathcal{I}$) abstraction of ϕ. Then ${}_\alpha T \models \phi^\alpha$ implies $T \models \phi$.*

Proof. The theorem is a corollary of Theorem 2, item 1 B, from [20]: Observe that $\tilde{\gamma}(\mathcal{I}^\alpha(p)) \subseteq \mathcal{I}(p)$ and that the consistency of α with $\mathcal{I}$ implies the consistency of $\tilde{\gamma}$ with $\mathcal{I}^\alpha$. $\square$

[4] Consistent abstraction corresponds to the notion of precise abstraction from [17].

Often it is more convenient to apply abstractions directly on the model M than on its transition system T. Such an abstraction on the level of M is well-developed within the *Abstract Interpretation* framework [8, 9, 12]. The requirement that Abstract Interpretation imposes on the relation between the concrete model M and its safe abstraction M^α can be formalized as a requirement on the relation between the data and the operations of the concrete system and their abstract counterparts as follows: Each value of the concrete domain Σ is mapped by a *description function* ρ_d into a value from the abstract domain ${}_\alpha\Sigma$. The abstract value "describes" the concrete value. We assume an ordering $\preceq$ on the abstract domain ${}_\alpha\Sigma$ according to the "precision" of abstract values: given a concrete value x and its abstract description $x^\alpha = \rho_d(x)$, we say that any $y^\alpha \in {}_\alpha\Sigma$ such that $x^\alpha \preceq y^\alpha$ is a *less precise* description of x.

For every operation (function) f on the concrete data domain, an abstract function f^α is defined, which "mimics" f. (For simplicity, we assume f to be a unary operation.) In general, the abstraction can be nondeterministic. This is formally captured by letting f^α be a function into the *powerset* over the domain of abstract values. The requirement of mimicking is then formally phrased with the following *safety statement*: $\forall x \in \Sigma\ \exists y \in f^\alpha(\rho_d(x)) : \rho_d(f(x)) \preceq y$.

A state s can be seen as a valuation vector $\langle v_0, v_1, \ldots, v_{n-1}\rangle$ and, thus, $S = \Sigma_0 \times \ldots \times \Sigma_{n-1}$, with $\Sigma_0, \ldots, \Sigma_{n-1}$ being the corresponding data domains. Assuming the same set of variables as in M, the state space of M^α is ${}_\alpha S = {}_\alpha\Sigma_0 \times \ldots \times {}_\alpha\Sigma_{n-1}$. We relate S and ${}_\alpha S$ via the description relation, which in our case is the function $\rho_s : S \to {}_\alpha S$ defined as $\rho_s(s) = \langle \rho_{d0}(v_0), \ldots, \rho_{d\,n-1}(v_{n-1})\rangle$, where $\rho_{d0}, \ldots, \rho_{d\,n-1}$ are description functions for the corresponding variables. (We assume a trivial (identity) mapping as description function for unabstracted variables.)

Let M^α be obtained by replacing each constant c and function f of M with their abstract versions, $T^\alpha = (S^\alpha, R^\alpha)$ be the transition system that corresponds to M^α, and let the safety statement hold for all functions. Obviously $S^\alpha = {}_\alpha\Sigma_0 \times \ldots \times {}_\alpha\Sigma_{n-1} = {}_\alpha S$. Moreover, for "usual" modelling languages, like PROMELA, $R^\alpha \supseteq {}_\alpha R$, which can be shown e.g. following 4.4.1 from [9]. This trivially implies ${}_\alpha T \sqsubseteq_{\alpha'} T^\alpha$, where α' is the identity function. Thus, by Theorem 1 a given formula ϕ^α is preserved from T^α via ${}_\alpha T$ to T, i.e., from M^α to M.

Timer Abstraction We employ the concept of timers to specify timing conditions imposed on the system. Each timer is related to a certain process and modelled by a timer variable. We denote the value of a timer t at a state s as $[\![t]\!]_s$. A timer can be activated by setting. Timer variables are mapped to integers; -1 represents a deactivated timer and larger values stand for active timers. A setting step $set(t, e)$ leads to the change of the timer value to the value given by expression e. A predicate $expire(t)$ is $true$ iff $[\![t]\!] = 0$. The transitions are assumed to be instantaneous. Time progression is modelled as a special transition called *tick* that decreases values of *all* active timers by 1 and leaves deactivated timers unmodified. Further, we refer to a segment of time separated by time progress steps as a *time slice*. We leave the semantics of time partially open here, since our approach does not depend on it. (We revisit this issue in Section 3.)

To prove that some property holds for all settings of a timer that are greater or equal to some value k, one often uses a timer abstraction similar to the one of [10]. For a timer t, the concrete domain of timer values $\Sigma = \mathbb{N} \cup \{-1\}$ is replaced with the abstract domain ${}_{\alpha}\Sigma_t = \{-1, 0, \ldots, k_t - 1, k_t^+\}$, where the value k_t is a positive value defined by the user assuming that the verification property still holds even if we do not distinguish between the values of the timer greater or equal to k_t. We overload the notation by using c $(-1 \leq c < k_t)$ as an abstract value representing the single concrete value i, while c^+ describes the set of concrete values $\{c, c+1, c+2, \ldots\}$.

The description function ρ_{dt} is defined as $\rho_{dt}(c) = c$, if $c < k_t$, and $\rho_{dt}(c) = k_t^+$, otherwise. Abstract operations on timers are defined in an intuitive way: setting a timer to value x becomes setting it to value $\rho_{dt}(x)$, $expire^{\alpha}(a)$ is *true* iff $a = 0$, and $tick^{\alpha}$ is a non-deterministic operation that changes the value of a timer from a to b according to the following rules: (1) if $a = -1$ then $b = -1$, (2) if $0 \leq a < k_t$ then $b = a - 1$ (where "$-$" works on abstract values as on integers), (3) if $a = x^+$ then $b \in \{x^+, x - 1\}$.

Lemma 2. *System M^{α} built from system M according to the rules given above is a safe abstraction of M.*

Proof. By a simple check that the safety statement is satisfied. $\square$

From now on we assume that systems under consideration have no deadlocks and infinite zero-time cycles (infinite traces with a finite number of *tick*'s). The absence of zero-time cycles can be checked on the abstract model by verifying the property $\Box\Diamond tick^{\alpha}$, which is a consistent abstraction of $\Box\Diamond tick$. The absence of deadlocks follows straightforward from the fact that time can progress even when no other action is possible in the system, and thus *tick* action is still possible.

Fair timer abstraction. An abstracted system contains more behavior than the original one. Therefore, positive verification results can be transferred from the abstract to the concrete system, while counterexamples can be spurious. Abstraction refinement is a common technique used in case spurious counterexamples are found (see e.g. [6]), though just a change of the granularity level does not always help—the sequence of refinements can turn out to be infinite.

Suppose we use the timer abstraction described above to prove that some property holds for all timer settings greater than or equal to some k_t. Due to the non-determinism introduced with the abstract version of *tick*, it becomes possible that the timer once set will never expire. That means that the states that are always reachable in the concrete system are not reached in the abstract system if $k_t^+ \xrightarrow{tick} k_t^+$ step is always chosen. Such a trace gives a spurious counterexample: In the concrete system the timer expires after a finite number of time slices. The only possible refinement is taking the same abstraction with a greater value of k. But the same trace where the timer never expires is still possible, so a counterexample would be produced again. Therefore, we need a different technique to cope with the problem.

Imposing a strong fairness condition that requires that for any trace where transition $k_t^+ \xrightarrow{tick} (k_t - 1)$ is infinitely often enabled it is infinitely often taken, gives incorrect results: One can easily build a (concrete) model where a timer t is infinitely often set to a new value (before it expires), so it can be seen every time as a new variable in the one-assignment setting. This observation leads us to the following definition of *t-fairness*:

Definition 3. *Given an LTS T of a system with a set of abstract timers $TVar^\alpha$, we say that a trace of T is* t-fair *iff for any $t \in TVar^\alpha$ the following holds: $k_t^+ \xrightarrow{tick} (k_t - 1)$ is infinitely often enabled implies that $k_t^+ \xrightarrow{tick} (k_t - 1)$ is infinitely often executed or $set(t, x)$, $x \in {}_\alpha\Sigma_t$, is infinitely often executed.*

This definition has a *strong fairness* pattern. Interestingly, due to the fact that the loop introduced on a timer with the abstraction is a *self-loop*, this requirement can be reformulated as a condition with a *weak fairness* pattern:

Lemma 4. *A trace ξ of T is* t-fair *iff for any $t \in TVar^\alpha$ the following holds: if there exists an infinite suffix σ of ξ such that $[\![t]\!]_{s_j} = k_t^+$ for every state of σ, then $set(t, k_t^+)$ is infinitely often executed along the trace.*

Proof. Let p, q, and r denote the propositions (from Def. 3) "$k_t^+ \xrightarrow{tick} (k_t - 1)$ is enabled", "$k_t^+ \xrightarrow{tick} (k_t - 1)$ is executed", and "$set(t, x)$, $x \in {}_\alpha\Sigma_t$, is executed", respectively.

$$\Box\Diamond p \to (\Box\Diamond q \vee \Box\Diamond r). \tag{1}$$

We can split the proposition r into a disjunction of two propositions r_1 and r_2: "$set(t, k_t^+)$ is executed" and "$set(t, x)$, where $x \neq k_t^+$, is executed", respectively. After straightforward transformations, (1) becomes

$$\neg(\Box\Diamond p \wedge \Diamond\Box(\neg q \wedge \neg r_1)) \vee \Box\Diamond r_2. \tag{2}$$

We will show that $\Box\Diamond p \wedge \Diamond\Box(\neg q \wedge \neg r_1)$ (*), is semantically equivalent to $\Diamond\Box p'$, where p' denotes the proposition "the value of t is k_t^+".

The conjunct $\Box\Diamond p$ says that $k_t^+ \xrightarrow{tick} (k_t - 1)$ is infinitely often enabled. Since we assume the absence of zero-time cycles, by the timer abstraction definition, this is equivalent to the proposition "timer t has value k_t^+ infinitely often". The conjunct $\Diamond\Box(\neg q \wedge \neg r_1)$ says that after some point in the execution sequence neither $k_t^+ \xrightarrow{tick} (k_t - 1)$ nor $set(t, x)$, with $x \neq k_t^+$, are executed. As these transitions are the only ones that can change the value of t from k_t^+ to a value different than k_t^+, we can conclude that from some point the value of t will remain k_t^+ forever.

For the other direction, we first observe that if t has value k_t^+ from some point on, then $k_t^+ \xrightarrow{tick} (k_t - 1)$ is enabled infinitely many times. (Again, we use the absence of zero-time cycles, i.e., a *tick* transition is executed infinitely often along any execution sequence.) Also, the other conjunct of (*) follows immediately: As $k_t^+ \xrightarrow{tick} (k_t - 1)$ and $set(t, x)$, where $x \neq k_t^+$, are the only statements which can change the value of the abstract timer t, they also cannot be executed after some point on.

Thus, we can replace (*) with the equivalent proposition $\Diamond\Box p'$ and rewrite (2) as $\Diamond\Box p' \to \Box\Diamond r_2$, which is the (weak t-fairness) condition of Lemma 4. $\Box$

Thus we can express the t-fairness criterion by the LTL formula $\Phi^\alpha = \bigwedge_{t \in TVar^\alpha}(\Diamond\Box p \to \Box\Diamond q)$, where p and q are propositions corresponding to the terms "$[\![t]\!]_{s_j} = k_t^+$" and "$set(t, k_t^+)$" from Lemma 4, respectively. Though Φ^α is formulated on states *and transitions*, it can be easily encoded as a property defined on the states of the system. (To express the fact that some transition q is infinitely often taken, one can e.g. extend the model with introducing a boolean variable b_q that is negated every time the transition is taken and replace $\Box\Diamond q$ with $\Box\Diamond b_q \wedge \Box\Diamond\neg b_q$.)

One can see the analogy between Φ^α and the definition of *weak fairness for processes*, where a timer set to k_t^+ corresponds to an enabled process and an execution of the *set* operation corresponds to an execution of an action by the process. Further, one can show that the t-fairness criterion Φ^α is a consistent abstraction of the LTL formula $\Phi = \bigwedge_{t \in TVar}(\Diamond\Box p' \to \Box\Diamond q')$, where p', q' are defined as "$[\![t]\!]_{s_j} \geq k_t$" and "$set(t, x)$, where $x \geq k_t$", respectively. This can be done by a simple check that p and q are consistent abstractions of p' and q', respectively. Indeed, let $s^\alpha \in \alpha(\{s\})$. Timer t has the value k_t^+ in the abstract state s^α iff t has a value greater than or equal to k_t in s. Similarly, t is set to some x which is greater than or equal to k_t by a transition which has s as the target state iff it is set by a transition in the abstract state which ends up in the state s^α with $[\![t]\!]_{s^\alpha} = k_t^+$.

Suppose we want to verify that $T \models \phi$ for some $\Box L_\mu^+$ (resp. $\Box L_\mu$) formula ϕ and a concrete system T without infinite zero-time traces. The "concrete" version of the abstract t-fairness condition, Φ, holds on any trace of T: If from some point on the value of timer t remains greater than or equal to k_t, then the timer must be infinitely often set to some value greater than k_t. Otherwise, since *tick* happens infinitely often, the value of t will eventually become less than k_t. Thus, $T \models \phi$ iff $T \models (\Phi \to \phi)$.

By Theorem 1 we know that instead of verifying $T \models (\Phi \to \phi)$ on the concrete system, we can verify its contracting (resp. consistent) abstraction $(\Phi \to \phi)^\alpha$ on the abstract system. By the definition of contracting (consistent) abstraction, the last formula is equivalent to $\Phi^\alpha \to \phi^\alpha$. In case ϕ does not refer to variables (timers) that are abstracted, the abstraction α is trivially a consistent abstraction for all atomic propositions in ϕ and we have $\phi^\alpha = \phi$. If ϕ does mention abstracted timers, one has to derive the contracting abstraction ϕ^α of ϕ. Finally, by Theorem 1, $T^\alpha \models (\Phi^\alpha \to \phi^\alpha)$ implies $T \models (\Phi \to \phi)$ and thus also $T \models \phi$.

Thus, by imposing t-fairness condition on the abstract model, we eliminate spurious counterexamples caused by unfair non-deterministic choices made by abstract functions.

3 Incorporating t-Fairness into the Verification Algorithm

To express the formula Φ^α as an LTL formula defined on the states of the system, one needs to introduce additional variables (see Section 2). Therefore it is computationally expensive to verify the formula $\Phi^\alpha \to \phi^\alpha$ and it is more convenient to incorporate the t-fairness requirement into the verification algorithm that verifies ϕ^α by considering t-fair traces only. In this section we describe how to embed the t-fairness check into a model-checking algorithm for LTL.

Since there is a strong analogy between *t-fairness* and *weak process fairness*, one can easily adapt any algorithm for model checking under weak process fairness. The algorithm we propose here is inspired by the weak process fairness algorithm used in SPIN [15, 2], which is a combination of the Nested Depth First Search (NDFS) algorithm [7] and Choueka's flag algorithm [5]. In the automata-theoretic approach, to verify a property expressed by an LTL formula, the negation of the formula is translated into a Büchi automaton, which is combined with the transition system representing the state space of the system. If the language accepted by the resulting automaton is empty, the property is satisfied. As a result, the model checking problem is reduced to a graph theoretic problem of finding *acceptance cycles*, i.e., cycles that contain states from a special designated set of accepting states. The absence of acceptance cycles means that the property holds for the system. Further on we assume that we work directly with the labelled transition system (LTS), which is the product of the Büchi automaton and the LTS of the system.

Given an LTS $T = (S, Act, \longrightarrow_T, s_{init}, F)$ of a composition of the transition system of a given abstract system with the Büchi automaton that represents the negation of a property to be verified, where S is a finite state space, Act is a set of actions, $\longrightarrow_T \subseteq S \times Act \times S$ is a transition relation, $s_{init} \in S$ is an initial state and $F \subseteq S$ is a set of accepting states. Our goal is to construct an extension of T that contains an acceptance cycle iff there exists a t-fair acceptance cycle in T. (We say that a cycle $s_0 \xrightarrow{a_0} ...s_n \xrightarrow{a_n} s_0$ is t-fair iff $\forall t \in TVar^\alpha$ there exists i, $(0 \leq i \leq n)$, such that $[\![t]\!]_{s_i} \neq k_t^+$ or $a_i = set(t, k_t^+)$.) Therefore, we will define this extension in such a way that any acceptance cycle would be t-fair by construction.

Let the abstract system have N abstract timers. Then we construct the extended LTS $T' = (S', Act', \longrightarrow_{T'}, s'_{init}, F')$ in the following way: The set of states of the extended system is a set of pairs (s, c), where $s \in S$ and $0 \leq c \leq N$. We call (s, c) a *c-replica* of s. (Note that not every replica (s, c) of a reachable state s of T will be reachable in T'). 0-replicas are the basic replicas of the states, while replicas $1, \ldots, N$ allow to track the behaviour of abstract timers $t_1, \ldots, t_N$, respectively. All the accepting states and the initial state of T' are 0-replicas of the accepting states and the initial state of T, respectively. All transitions from accepting states lead to 1-replicas only. Transitions from a c-replica (s, c), related to timer t_c, lead either to the c-replicas, or, when they guarantee t-fair behaviour w.r.t. timer t_c, to the next ((c+1) mod $(N+1)$) replica. Since all the acceptance states are 0-*replicas*, any acceptance cycle contains for every abstract timer at least one transition that guarantees t-fairness.

The verification algorithm starts the construction of T' from the initial state $(s_{init}, 0)$ and proceeds by adding the 0-replicas in accordance with the transition function $\longrightarrow_T$ until an accepting state is met. If an accepting state s is encountered, the algorithm adds a dummy τ-step that connects the 0-replica of s with the 1-replica of the same state. A move from a c-replica with $1 \leq c \leq N$ to the $((c+1) \mod (N+1))$-replica happens when a state is encountered in which t_c has a value different from k_t^+ or a step setting timer t_c is taken, i.e. *when the t-fairness condition for t_c is fulfilled.* (A move from a 0-replica to a 1-replica is possible only by τ-steps connecting the replicas of the same accepting state.) For the rest, the algorithm adds states following the transition function $\longrightarrow_T$.

Theorem 5. *Given an LTS $T = (S, Act, \longrightarrow_T, s_{init}, F)$ with abstract timers $t_1, \ldots, t_N$ and its smallest extension $T' = (S', Act', \longrightarrow_{T'}, s'_{init}, F')$ that satisfies the following conditions:*

1. *$Act' = Act \cup \{\tau\}$;*
2. *$s'_{init} = (s_{init}, 0)$;*
3. *$(s, 0) \xrightarrow{a}_{T'} (s_1, 0)$ if $(s, 0) \in S'$ and $s \xrightarrow{a}_T s_1$ and $s \notin F$;*
4. *$(s, 0) \xrightarrow{\tau}_{T'} (s, 1)$ if $(s, 0) \in S'$ and $s \in F$;*
5. *$(s, c) \xrightarrow{a}_{T'} (s_1, c_1)$ if $(s, c) \in S'$ and $c > 0$ and $s \xrightarrow{a}_T s_1$ with $c_1 = ((c+1) \mod (N+1))$ if $([\![t_c]\!]_s \neq k_{t_c}^+$ or $a = set(t_c, k_{t_c}^+))$, and $c_1 = c$ otherwise;*
6. *$F' = S' \cap \{(s, 0) \mid s \in F\}$.*

Then the following statements hold:

1. *$(S, Act, \longrightarrow_T, s_{init})$ and $(S', Act', \longrightarrow_{T'}, s'_{init})$ are branching bisimilar.*
2. *T contains a reachable* t-fair *acceptance cycle iff T' contains a reachable acceptance cycle.*

Proof. 1. Consider $Q \subseteq S \times S'$ where $(s, s') \in Q$ iff $s' = (s, c)$ where $0 \leq c \leq N$. It is straightforward to check by case analysis that Q is a weak bisimulation. Since system T is τ-free, T and T' are branching bisimilar [24].

2. Notice that all acceptance cycles of the extended state space are t-fair by construction: An acceptance cycle contains at least one accepting state; this state is a 0-replica and has outgoing transitions to 1-replicas only. As transitions from a c-replica lead either to c-replicas, or to "neighbour" $((c+1) \mod (N+1))$-replicas $(0 \leq c \leq N)$, for any c, the cycle includes a c-replica $(s, c), s \in S$. Every move from a c-replica to its neighbour satisfies the t-fairness condition for timer t_c, so for every abstract timer there is a transition in the cycle satisfying the t-fairness condition and thus the cycle is t-fair.

Due to the bisimulation result, any acceptance cycle of T' (which is always t-fair) has a corresponding t-fair acceptance cycle in T.

In the opposite direction: Assume that there is a trace $s_{init} \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ in T that contains a fair acceptance cycle. Then there are s_i, s_j such that $s_i = s_j$ with $j > i$. The path π from s_i to s_j contains at most $m = (j-i)$ distinct states. Trace $\sigma = s_{init} \xrightarrow{a} ...s_i....s_i...s_i$ going through the cycle $N+1$ times is also the valid trace of T. Due to the bisimulation result, there is a trace σ' in T' that

Procedure 6 **(dfs**(s, c)**).**

add (s, c) to S'	add a pair to the state space
if $c = 0$ and $s \in F$	0-replica and state s is accepting
then if $(s, 1) \notin S'$ then $dfs(s, 1)$;	τ-step from 0-replica to 1-replica
else	
for all $s \xrightarrow{a}_T s_1$ do	for all transitions enabled in s
if $c > 0$ and $(a = set(t_c, k^+_{t_c})$ or $[\![t_c]\!]_s \neq k^+_{t_c})$	t-fairness condition
then $c_1 = (c + 1) \bmod N$	the next replica number
else $c_1 = c$;	the same replica number
if $(s_1, c_1) \notin S'$ then $dfs(s_1, c_1)$;	recursive call
od;	

Fig. 2. Generating t-fair extension of S

mimics σ. The suffix ξ' of σ' that mimics passing through the cycle $N + 1$ times contains at least $m(N + 1) + 1$ states. The states of ξ' are replicas of the states of π, therefore at most $m(N + 1)$ of them are distinct. Thus, there is at least one state that is present in ξ' twice, and ξ' is a cycle.

Now we shall show that ξ' is an *acceptance* cycle. We denote the suffix of σ corresponding to ξ' as ξ and pick up an arbitrary state s of ξ. Then ξ' contains some state (s, c), $0 \leq c \leq N$. Assume that $c > 0$. Since ξ is a t-fair cycle, there are some states q_1, q_2 reachable from s such that $q_1 \xrightarrow{a}_T q_2$ and ($[\![t_c]\!]_{q_1} \neq k^+_t$ or $a = set(t_c, k^+_t)$). Hence there exists a transition from the c-replica q_1 to the $((c + 1) \mod N)$-replica q_2 in ξ'. Proceeding in the same way, we will obtain transitions leading to some $((c + 2) \mod N)$-replica, etc., and eventually we arrive at a 0-replica. Thus, we conclude that ξ' contains at least one 0-replica of some state. In T', transitions from 0-replicas of non-accepting states lead to 0-replicas, and transitions from 0-replicas of accepting states lead to 1-replicas. Since ξ contains an accepting state, due to the bisimulation result, ξ' contains an accepting state as well and thus it is an accepting cycle of T'. $\square$

We call the extension T' a *t-fair extension* of T. An algorithm that generates the extended state space in a depth first search (DFS) manner is given in Fig. 2. It is straightforward to prove the following claim:

Lemma 7. *Given an LTS T, let T' be a system produced from system T by applying Procedure 6. Then T' is a t-fair extension of T.*

To detect acceptance cycles, DFS is extended with a cycle-check procedure (Fig. 3). Whenever Procedure 8 detects an accepting state, it starts Procedure 9, which is again a DFS, that reports an accepting state if the seed state is matched within the cycle-check. Here we omit a detailed description of the NDFS algorithm and refer the interested reader to [7].

The correctness of the algorithm is given by the following claim:

Procedure 8 **($\mathbf{ndfs}_1(s,\ c)$).**

add $(s, c, 0)$ to S' — add a pair to the state space
if $c = 0$ and $s \in F$ — 0-replica, and state s is accepting
then if $(s, 1, 0) \notin S'$ then $ndfs_1(s, 1)$; — τ-step from 0-replica to 1-replica
else
  for all $s \xrightarrow{a}_T s_1$ do — for all transitions enabled in s
    if $c > 0$ and ($a = set(t_c, k^+_{t_c})$ or $[\![t_c]\!]_s \neq k^+_{t_c}$) — t-fairness condition
    then $c_1 = (c + 1) \bmod N$ — the next replica number
    else $c_1 = c$; — the same replica number
    if $(s_1, c_1, 0) \notin S'$ then $ndfs_1(s_1, c_1)$; — recursive call
  od;
if $c = 0$ and $s \in F$ then $seed := (s, 0, 1)$; $ndfs_2(s, 0)$; — set the seed and start ndfs_2

Procedure 9 **($\mathbf{ndfs}_2(s,\ c)$).**

add $(s, c, 1)$ to S' — add a pair to the state space
if $c = 0$ and $s \in F$ — 0-replica, and state s is accepting
then if $(s, 1, 1) \notin S'$ then $ndfs_2(s, 1)$; — τ-step from 0-replica to 1-replica
else
  for all $s \xrightarrow{a}_T s_1$ do — for all transitions enabled in s
    if $c > 0$ and ($a = set(t_c, k^+_{t_c})$ or $[\![t_c]\!]_s \neq k^+_{t_c}$) — t-fairness condition
    then $c_1 = (c + 1) \bmod N$ — the next replica number
    else $c_1 = c$; — the same replica number
    if $seed = (s, c_1, 1)$ then REPORT CYCLE! — seed is matched, report the cycle
    else if $(s_1, c_1, 1) \notin S'$ then $ndfs_2(s_1, c_1)$; — recursive call
  od;

Fig. 3. NDFS version of Procedure 6

Theorem 10. *Given an LTS T, Procedure 8 called with $(s_{init}, 0)$ reports an acceptance cycle iff there exists a reachable t-fair acceptance cycle in T.*

Proof. Follows from the correctness of the NDFS algorithm from [7] by observing that the algorithm is actually NDFS from [7] applied on the extended state space T'. □

The last result completes the series of claims that guarantee the soundness of the verification approach proposed in this paper. If no acceptance cycle is detected then the verified property holds for t-fair traces of the abstract system and therefore also for the concrete system.

Time complexity of the NDFS Algorithm in Fig. 3 is $O(N \cdot |T|)$, where N is the number of timers, while $|T|$ is the size (states and transitions) of the abstract system state space. Memory space needed to save T' is virtually the same as the one for T. Instead of keeping each of the N replicas (s, i), $(1 \leq i \leq N)$ one can save only the "useful" part s plus additional $2(N + 1)$ bits, like it is done for process fairness in SPIN. The first $N+1$ bits correspond to the replicas in the main depth first search of the NDFS algorithm, while the second group of $(N+1)$ bits corresponds to the nested DFS. If bit i of the first group is set then this means that the state (s,i) has been visited by the algorithm. Similarly for the second group. As the description of s is usually much greater than $2(N+1)$ bits, the bookkeeping overhead is negligible.

4 *T*-fairness in DTSpin

DTSPIN [3] is a discrete-time extension of SPIN [15] that has all verification features of SPIN. It was successfully applied for debugging and verification of timed models of industrial size protocols (see e.g. [4, 16]). DTSPIN is designed for the verification of systems where delays are significantly larger than the duration of the events within the system. Therefore, system transitions are assumed to be instantaneous. DTSPIN employs the concept of timers to express time aspects of a system. In DTPROMELA, the input language of DTSPIN, timers are modelled by variables of a predefined type *timer*. The data domain and the operations on timers are defined as in Section 2.

Since the system transitions are assumed to be instantaneous, time progress has the least priority in the system and may take place only when the system is *blocked*. A special process *Timer* ticks all the active timers down in case the system is blocked. DTSPIN employs PROMELA's statement *timeout* to check whether the system is blocked. To ensure that time progression has the least priority, the usage of *timeout* is reserved for the implementation of time progression and forbidden in DTPROMELA specifications. Note that by the definition of *tick*, all DTPROMELA models are *deadlock-free*.

To implement the timer abstraction defined in Section 2, we extend DTPROMELA with a new data type $timer^{\alpha}$ for abstract timers and define the operations on them as macros. The abstract version of *tick*, $tick^{\alpha}$, decreases values of active abstract timers if they are different from k_t^+. If a timer has the k_t^+ value, the non-deterministic choice is made between decreasing the value of the timer to $(k_t - 1)$ and leaving it unmodified. Our fairness algorithm from Section 3 is implemented by means of a PAN2TFPAN Java program that transforms the *pan* verifier generated by SPIN for the verification of the property without t-fairness into a new one that checks the property under t-fairness. The transformation is automatic and does not require any interaction with the user.

The user applies thus the following scheme for the verification: (1) Choose timers of a concrete model that should be abstracted and define a k_t value for each of those timers; (2) Redefine the type of the chosen times to $timer^{\alpha}$ and redefine the *set* operations according to the k_t values; (3) Check whether the abstract system is free from zero-time cycles, i.e. check whether *tick* happens infinitely often. This is done by checking LTL formula: $\Box\Diamond timeout$. (In DTSPIN, time progresses if the statement *timeout* of PROMELA is *true*. Since this statement is forbidden to use in DTPROMELA specifications, $\Box\Diamond timeout$ expresses the absence of zero-time cycles.) (4) Formulate the abstract version of the property to check and generate the *pan* verifier for this property; (5) Transform the *pan* verifier with PAN2TFPAN to the new *pan* verifier, which will check the property under the t-fairness condition. Positive verification results imply that the property holds for the concrete system as well. If the property gets violated on the abstract system, the counterexample is generated, and the user checks whether the counterexample is spurious or not.

5 Experimental results

In this section we describe some experimental results that show the efficiency of our approach. Our test cases are the positive acknowledgment retransmission protocol (PAR) [23] and Fischers mutual exclusion protocol [19]. We compare the results obtained when we specify t-fairness as LTL formulas according to strong fairness and weak fairness patterns (we will refer to it as verifying with strong/weak fairness respectively) with the results obtained with our prototype implementation of the algorithm from Section 3 in DTSPIN, which we refer to as built-in t-fairness. Our prime goal here is to compare the performance of the three methods rather than to verify the protocols.

Experiments with the Positive Acknowledgment Retransmission Protocol (PAR) PAR [23] is a classical example of a communication protocol where time issues are essential for the correct functionality of the protocol. PAR involves a sender, a receiver, a message channel and an acknowledgment channel. The sender receives a frame from the upper layer, sends it to the receiver via the message channel and waits for a positive acknowledgment from the receiver via acknowledgment channel. When the receiver delivered the message to the upper layer, it sends the acknowledgment to the sender. After the positive acknowledgment is received, the sender becomes ready to send the next message. The channels delay the delivery of messages. Moreover, they can lose or corrupt messages. Therefore, the sender handles lost frames by timing out. If the sender times out, it re-sends the message. As known, the protocol functions correctly only under the following condition: the timeout of sender should be greater than the sum of delays on channels.

We specified PAR in DTPROMELA using concrete timers to represent delays on the channels and the sender timeout. Our goal was to check that if the channels do not lose messages continuously, no message reordering occurs and no message gets lost, under condition that the timeout of the sender is greater than the sum of the (given) delays on the channels. To prove the property for an arbitrary message sequence we used a well-known canonical abstraction [14, 25] and defined two abstract environment processes: one representing an upper layer for the sender and another one for the upper layer of the receiver. Then we abstracted the sender's timer to check the property for *all* values greater than the sum of the channels' delays.

Without t-fairness, the property gets violated, since there exists a trace where the abstract timer of the sender never expires, staying in the loop $k_t^+ \xrightarrow{tick} k_t^+$

Table 1. PAR

pattern	states	transitions	memory(Mb)	time
strong fairness	825761	5.10962e+06	52.286	0:21.00
weak fairness	227569	1.49527e+06	15.320	0:05.98
built-in t-fairness	100275	390012	6.693	0:01.56

(we obtained a t-unfair trace as counterexample). Under the t-fairness condition, we proved that the property holds. Table 1 contains information on the time and memory consumption for the verification with DTSpin of the property formulated with the strong and weak fairness patterns and for the verifier with built-in t-fairness.

Fischer's mutual exclusion protocol Our second test example is Fischer's mutual exclusion protocol. The protocol uses time constraints and a shared variable to ensure mutual exclusion in a system that consists of N processes running in parallel and competing for a critical section. We assume that each process has a unique *id* from 1 to N. The initial value of the shared variable x is 0. When a process observes that x is 0, it waits for *at most* δ_1 time units and then writes its *id* to x. After that, it waits for *at least* δ_2 time units, and if x still equals the process *id*, the process enters the critical section. The process stays in the critical section for some time and then leaves it.

We have specified Fischer's mutual exclusion protocol in DTPromela using concrete timers to represent delays not larger than δ_1 and abstract timers to represent delays which are at least δ_2. As known, mutual exclusion is ensured provided that $\delta_1 < \delta_2$. We have checked the property that if there comes a request of access to the critical section, one of the processes will get it. Table 2 contains results for strong, weak and built-in t-fairness for the case of two, three and four processes. Note that the number of abstracted timers in this example is equal to the number of processes.

Table 2. Fischer's mutual exclusion

pattern	num. of proc.	states	transitions	memory(Mb)	time
strong fairness	2	41384	171586	4.363	0:00.46
weak fairness	2	4705	13053	2.724	0:00.08
built-in t-fairness	2	1236	4181	1.573	0:00.01
strong fairness	3	3.28599e+06	2.01406e+07	190.539	1:01.79
weak fairness	3	115874	362068	8.561	0:01.22
built-in t-fairness	3	21592	110332	2.700	0:00.26
strong fairness	4	out of memory			
weak fairness	4	2.60665e+06	9.2549e+06	151.729	0:38.34
built-in t-fairness	4	346903	2.45733e+06	20.927	0:05.69

The experiments were done on AMD Athlon(TM) XP 2400+ with 1Gb of memory. In all experiments, the verification with built-in t-fairness took significantly less time and memory than the verification with strong and weak fairness patterns expressed as LTL formulas. The prototype implementation PAN2TFPAN and the models can be found at `www.cwi.nl/~ustin/tfair.html`.

6 Conclusion

In this paper we considered a timer abstraction that introduces a cyclic behavior on abstract timers that is not present at the concrete level. This could lead to spurious counterexamples for liveness properties. We showed how one can eliminate those by imposing a strong fairness constraint on the traces of the abstract model. Using the fact that the loop on the abstract timer is a self-loop for this abstract timer (though there is possibly no self-loop on the corresponding LTS), we transformed the strong fairness constraint into a constraint which has a weak fairness pattern, and embedded it into the verification algorithm. Our experiments with the prototype implementation of the algorithm were encouraging. We conjecture that the ideas in this paper can also be used for other data abstractions that introduce self-loops on the abstracted data.

References

1. K. Baukus, Y. Lakhnech, and K. Stahl. Verification of parameterized protocols. *Journal of Universal Computer Science*, 7(2):141–158, 2001.
2. D. Bošnački. Partial-order reduction in presence of rendez-vous communication with unless constructs and weak fairness. In *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th Int. SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*. Spriner-Verlag, 1999.
3. D. Bošnački and D. Dams. Integrating real time into Spin: A prototype implementation. In *Proc. of Formal Description Techniques and Protocol Specification, Testing, and Verification*. Kluwer Academic Publishers, 1998.
4. D. Bošnački, D. Dams, L. Holenderski, and N. Sidorova. Verifying SDL in Spin. In *TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
5. Y. Choueka. Theories of automata on ω-tapes: a simplified approach. *Journal of Computer and System Science*, 8:117–141, 1974.
6. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Int. Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*. Springer, 2000.
7. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.
8. P. Cousot and R. Cousot. Abstract interpretaion: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL '73*. ACM, January 1973.
9. D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD dissertation, Eindhoven University of Thechnology, July 1996.

10. D. Dams and R. Gerth. The bounded retransmission protocol revisited. *Electronic Notes in Theoretical Computer Science*, 9, 1999.
11. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems: Abstraction preserving $\forall$CTL*, $\exists$CTL*, and CTL*. In *Proc. of PROCOMET '94*. IFIP, North-Holland, June 1994.
12. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2), 1997.
13. M. M. Gallardo, J. Martine, P. Merino, and E. Pimentel. αSpin: Extending Spin with abstraction. In *9th Int. SPIN Workshop, Grenoble, France 2002*, volume 2318 of *Lecture Notes in Computer Science*, pages 254–258, 2002.
14. S. Graf. Verification of a distributed cache memory by using abstractions. In *Workshop on Computer-Aided Verification, CAV'94, Stanford*. LNCS 818, Springer Verlag, 1994.
15. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.
16. N. Ioustinova, N. Sidorova, and M. Steffen. Closing open SDL-systems for model checking with DT Spin. In *Proc. of Formal Methods Europe (FME'02)*, volume 2391 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
17. Y. Kesten and A. Pnueli. Control and data abstraction: The cornerstones of practical formal verification. *Int. Journal on Software Tools for Technology Transfer*, 2(4):328–342, 2000.
18. Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.
19. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions in Computer Systems*, 5(1):1–11, 1987.
20. 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.
21. B. D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs I. *Acta Informatica*, 21:125–169, 1984.
22. A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In *Computer Aided Verification : 14th Int. Conference, CAV 2002, Copenhagen, Denmark, July 27-31, 2002. Proc.*, volume 2404 of *Lecture Notes in Computer Science*, pages 107 – 122. Springer, 2002.
23. A. S. Tanenbaum. *Computer Networks*. Prentice Hall Int. Inc., 1981.
24. R. J. van Glabbeek and R. P. Weijland. Branching time and abstraction in bisimulation semantics. In *Proc. IFIP'89*, pages 613–618. North-Holland, 1989.
25. P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 184–192, St. Petersburgh, January 1986.