Abstracting and Counting Synchronizing Processes

Zeinab Ganjei, Ahmed Rezine*, Petru Eles, and Zebo Peng

Linköping University, Sweden

Abstract. We address the problem of automatically establishing synchronization dependent correctness (e.g. due to using barriers or ensuring absence of deadlocks) of programs generating an arbitrary number of concurrent processes and manipulating variables ranging over an infinite domain. This is beyond the capabilities of current automatic verification techniques. For this purpose, we define an original logic that mixes variables refering to the number of processes satisfying certain properties and variables directly manipulated by the concurrent processes. We then combine existing works on counter, predicate, and constrained monotonic abstraction and build an original nested counter example based refinement scheme for establishing correctness (expressed as non reachability of configurations satisfying formulas in our logic). We have implemented a tool (PACMAN, for predicated constrained monotonic abstraction) and used it to perform parameterized verification for several programs whose correctness crucially depends on precisely capturing the number of processes synchronizing using shared variables.

Key words: parameterized verification, counting logic, barrier synchronization, deadlock freedom, multithreaded programs, counter abstraction, predicate abstraction, constrained monotonic abstraction

1 Introduction

We address the problem of automatic and parameterized verification for concurrent multithreaded programs. We focus on synchronization related correctness as in the usage of barriers or integer shared variables for counting the number of processes at different stages of the computation. Such synchronizations orchestrate the different phases of the executions of possibly arbitrary many processes spawned during runs of multithreaded programs. Correctness is stated in terms of a new counting logic that we introduce. The counting logic makes it possible to express statements about program variables and variables counting the number of processes satisfying some properties on the program variables. Such statements can capture both individual properties, such as assertion violations, and global properties such as deadlocks or relations between the numbers of processes satisfying certain properties.

* In part supported by the 12.04 CENIIT project.

Synchronization among concurrent processes is central to the correctness of many shared memory based concurrent programs. This is particularly true in certain applications such as scientific computing where a number of processes, parameterized by the size of the problem or the number of cores, is spawned in order to perform heavy computations in phases. For this reason, when not implemented individually using shared variables, constructs such as (dynamic) barriers are made available in mainstream libraries and programming languages such as Pthreads, java.util.concurrent or OpenMP.

Automatically taking into account the different phases by which arbitrary many processes can pass is beyond the capabilities of current automatic verification techniques. Indeed, and as an example, handling programs with barriers of arbitrary sizes is a non trivial task even in the case where all processes only manipulate boolean variables. To enforce the correct behaviour of a barrier, a verification procedure needs to capture relations between the number of processes satisfying certain properties, for instance that there are no processes that are not waiting for the barrier before any process can cross it. This amounts to testing that the number of processes at certain locations is zero. Checking violations of program assertions is then tantamount to checking state reachability of a counter machine where counters track the number of processes satisfying predicates such as being at some program location. No sound verification techniques can therefore be complete for such systems.

Our approach to get around this problem builds on the following observation. In case there are no tests for the number of processes satisfying certain properties (e.g. being in specific programs locations for barriers), symmetric boolean concurrent programs can be exactly encoded as counter machines without tests, i.e., essentially vector addition systems (VASS). For such systems, state reachability can be decided using a backwards exploration that only manipulates sets that are upward closed with respect to the component wise ordering [2, 7]. The approach is exact because of monotonicity of the induced transition system (more processes can fire more transitions since there are no tests on the numbers of processes). Termination is guaranteed by well quasi ordering of the component wise ordering on the natural numbers. The induced transition system is no more monotonic in the presence of tests on the number of processes. The idea in monotonic abstraction [10] is to modify the semantics of the entailed tests (e.g., tests for zero for barriers), such that processes not satisfying the tests are removed (e.g., tests for zero are replaced by resets). This results in a monotonic over-approximation of the original transition system and spurious traces are possible. This is also true for verification approaches that generate (broadcast) concurrent boolean programs as abstractions of concurrent programs manipulating integer variables. Such boolean approximations are monotonic even when the original program (before abstraction) can encode tests on the number of processes and is therefore not monotonic. Indeed, having more processes while respecting important relations between their numbers and certain variables in the original programs does not necessarily allow to fire more transitions (which is what abstracted programs do in such approaches).

Our approach consists in two nested counter example guided abstraction refinement loops. We summarize our contributions in the following points.

1. We propose an original *counting logic* that allows to express statements about program variables and about the number of processes satisfying certain predicates on the program variables.
2. We implement the outer loop by leveraging on existing symmetric predicate abstraction techniques. We encode resulting boolean programs in terms of a monotonic counter machine where reachability of configurations satisfying a *counting property* from our logic is captured as a state reachability problem.
3. We explain how to strengthen the counter machine using *counting invariants*, i.e. properties from our logic that hold on all runs. these can be automatically generated using classical thread modular analysis techniques.
4. We leverage on existing constrained monotonic abstraction techniques to implement the inner loop and to address the state reachability problem.
5. We have implemented both loops, together with automatic counting invariants generation, in a prototype (Pacman) that allowed us to automatically establish or refute counting properties such as deadlock freedom and assertions. All programs we report on may generate arbitrary many processes.

Related work . Several works consider parameterized verification for concurrent programs. In [9] the authors use counter abstraction and truncate the counters to obtain a finite state system. Environment abstraction [4] combines predicate and counter abstraction. Both [9, 4] can require considerable interaction and human ingenuity to find the right predicates. The works in [8, 1] explore finite instances and automatically check for cutoff conditions. Except for checking larger instances, it is unclear how to refine entailed abstractions. The closest works to ours are [10, 3, 5]. We introduced (constrained) monotonic abstraction in [10, 3]. Monotonic abstraction was not combined with predicate abstraction, nor did it explicitly target counting properties or dynamic barrier based synchronization. In [5], the authors propose a predicate abstraction framework for concurrent multithreaded programs. As explained earlier such abstractions cannot exclude behaviours forbidden by synchronization mechanisms such as barriers. In our work, we build on [5] in order to handle shared and local integer variables. To the best of our knowledge, our work is the first automatic verification approach that specifically targets parameterized correctness of programs involving constructs where the number of processes are kept track of (e.g, using barriers).

Outline. We start by illustrating our approach using an example in Sec. 2 and introduce some preliminaries in Sec. 3. We then define concurrent programs and describe our counting logic in Sec. 4. Next, we explain the different phases of our nested loop in Sec. 5 and report on our experimental results in Sec. 6. We finally conclude in Sec. 7.

2 A Motivating Example

Consider the concurrent program described in Fig. 1. In this example, a *main* process spawns (transition t_1) an arbitrary number (*count*) of *proc* processes (at location $proc@lc_{ent}$). All processes share four integer variables (namely *max*, *prev*, *wait* and *count*) and a single boolean variable *proceed*. Initially, the variables *wait* and *count* are 0 while *proceed* is false. The other variables may assume any value belonging to their respective domains. Each *proc* process possesses a local integer variable *val* that can only be read or written by its owner. Each *proc* process assigns to *max* the value of its local variable *val* (may be any integer value) in case the later is larger than the former. Transitions t_6 and t_7 essentially implement a barrier in the sense that all *proc* processes must have reached $proc@lc_3$ in order for any of them to move to location $proc@lc_4$. After the barrier, the *max* value should be larger or equal to any previous local *val* value stored in the shared *prev* (i.e., $prev \leq max$ should hold). Violation of this assertion can be captured with the *countingpredicate* (introduced in Sec 4) $(proc@lc_4 \wedge \neg(prev \leq max))^{\#} \geq 1$ stating that the number of processes at location $proc@lc_4$ and witnessing that $prev > max$ is larger or equal than 1.

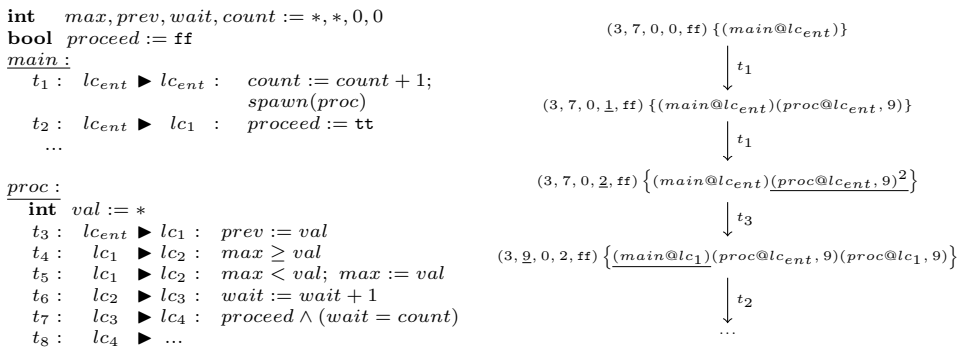

Fig. 1. The max example (left) and a possible run (right). The assertion $(proc@lc_4 \wedge \neg(prev \leq max))^{\#} \geq 1$ cannot be violated when starting with a single *main* process.

A possible run of the concurrent program is depicted to the right of Fig. 1. Observe that we write $(proc@lc_{ent}, 9)^2$ to mean that there are two *proc* processes at location lc_{ent} s.t. their local *val* are both equal to 9. In other words, we use counter abstraction since the processes are symmetric, i.e., processes running the same lines of code with equal local variables are interchangeable and we do not need to differentiate them. The initial configuration of this run is given in terms of the values of $max, prev, wait, count$ and *proceed*, here $(3, 7, 0, 0, \texttt{ff})$, and of the *main* process being at location lc_{ent}. The *main* process starts by incrementing the *count* variable and by spawning a *proc* process twice.

The assertion $(proc@lc_4 \wedge \neg(prev \leq max))^{\#} \geq 1$ is never violated under any run starting from a single main process. In order to establish this fact, any verification procedure needs to take into account the barrier in t_7 in addition to the two sources of infinitness; namely, the infinite domain of the shared and local variables and the number of *proc*s that may participate in the run. Until now, the closest works to ours deal with these two sources of infinitness separately and cannot capture facts that relate them, namely, the values of the program variables and the number of generated processes. Any sound analysis that does not take into account that the *count* variable holds the number of spawned *proc* processes and that *wait* represents the number of *proc* processes at locations lc_3 or later will not be able to discard scenarios were a *proc* process executes $prev := val$ although one of them is at $proc@lc_4$. Such an analysis will therefore fail to show that $prev \leq max$ each time a process is at line $proc@lc_4$.

Our original nested CEGAR loop, called **P**redic**a**ted **C**onstrained **M**onotonic **A**bstractio**n** and depicted in Fig. 2, systematically leverages on simple facts that relate numbers of processes to the variables manipulated in the program. This allows us to verify or refute safety properties (e.g., assertions, deadlock freedom) depending on complex behaviors induced by constructs such as dynamic barriers. We illustrate our approach in the remaining on the max example of Fig. 1.

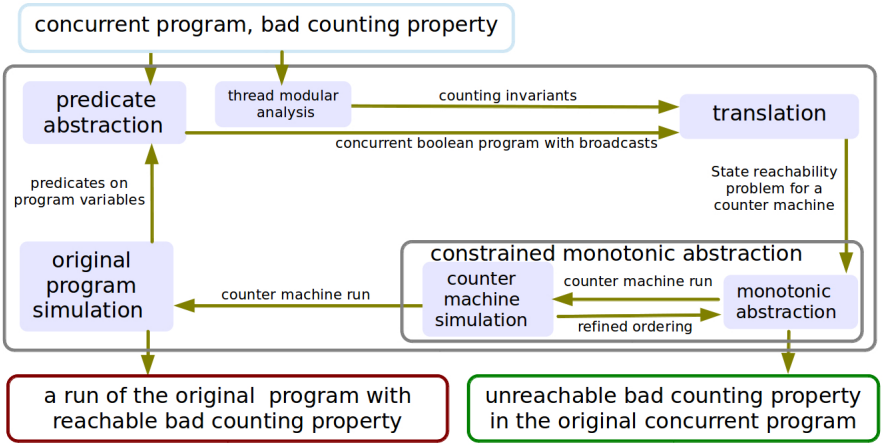

Fig. 2. **P**redic**a**ted **C**onstrained **M**onotonic **A**bstractio**n**

From concurrent programs to boolean concurrent programs. We build on the recent predicate abstraction techniques for concurrent programs. Such techniques would discard all variables and predicates and only keep the control flow together with the *spawn* and *join* statements. This leads to a number of counter example guided abstraction refinement steps (the outer CEGAR loop in Fig. 2) that require the addition of new predicates. Our implementation adds the

predicates *proceed*, $prev \leq val$, $prev \leq max$, $wait \leq count$, $count \leq wait$. It is worth noticing that all variables of the obtained concurrent program are finite state (in fact booleans). Hence, one would need a finite number of counters in order to faithfully capture the behavior of the abstracted program using counter abstraction. In addition, some of the transitions of the program, such as t_3 where a shared variable is updated, are abstracted using a broadcast transition.

From concurrent boolean programs to counter machines. Given a concurrent boolean program, we generate a counter machine that essentially boils down to a vector addition system with transfers. Each counter in the machine counts the number of processes at some location and with some local variables combination. One state in the counter machine represents reaching a configuration satisfying the counting property we want to verify. The other states correspond to the possible combinations of the global variables. The transfers represent broadcast transitions. Such a machine cannot relate the number of processes in certain locations (for instance the number of spawned processes *proc* so far) to the predicates that are valid at certain states (for instance that $count = wait$). In order to remedy to this fact, we make use of *counting invariants* that relate program variables, *count* and *wait* in the following invariants, to the number of processes at certain locations.

$$count = (proc@lc_{ent})^{\#} + \sum_{i \geq 1} (proc@lc_i)^{\#} \qquad wait = \sum_{i \geq 3} (proc@lc_i)^{\#}$$

We automatically generate such invariants using a simple thread modular analysis that tracks the number of processes at each location. Given such counting invariants, we constrain the counter machine and generate a more precise machine for which state reachability may now be undecidable.

Constrained monotonic abstraction. We monotonically abstract the resulting counter machine in order to answer the state reachability problem. Spurious traces are now possible. For instance, we remove processes violating the constraint imposed by the barrier in Fig.1. This example illustrates a situation where such approximations yield false positives. To see this, suppose two processes are spawned and *proceed* is set to `tt`. A first process gets to lc_3 and waits. The second process moves to lc_1. Removing the second process opens the barrier for the first process. However, the assertion can now be violated because the removed process did not have time to update the variable max. Constrained monotonic abstraction eliminates spurious traces by refining the preorder used in monotonic abstraction. For the example of Fig.1, if the number of processes at lc_1 is zero, then closing upwards will not alter this fact. By doing so, the process that was removed in forward at lc_1 is not allowed to be there to start with, and the assertion is automatically established for any number of processes. The inner loop of our approach can automatically add more elaborate refinements such as comparing the number of processes at different locations. Exact traces of the counter machine are sent to the next step and unreachability of the control location establishes safety of the concurrent program.

Trace Simulation. Traces obtained in the counter machine are real traces as far as the concurrent boolean program is concerned. Those traces can be simulated on the original program to find new predicates (e.g., using Craig interpolation) and use them in the next iteration of the outer loop.

3 Preliminaries

We write $\mathbb{N}$ to mean the set of natural numbers and $\mathbb{Z}$ to mean the one of integer numbers. We write k to mean a constant in $\mathbb{Z}$ and b to mean a boolean value in $\{\mathtt{tt}, \mathtt{ff}\}$. We use v, s, l, c, a to mean integer variables and $\tilde{v}, \tilde{s}, \tilde{l}$ to mean boolean variables. We let V, S, L, C and A (resp. $\tilde{V}, \tilde{S}$ and $\tilde{L}$) denote sets of integer variables (resp. sets of boolean variables). We let $\sim$ be an element in $\{<, \leq, =, \geq, >\}$. An expression e (resp. predicate π) belonging to the set $\mathtt{exprsOf}(V)$ (resp. $\mathtt{predsOf}(\tilde{V}, E)$) of arithmetic expressions (resp. boolean predicates) over integer variables V (resp. boolean variables $\tilde{V}$ and arithmetic expressions E) is defined as follows.

$$\begin{array}{lr} e ::= k \mid v \mid (e \ + \ e) \mid (e \ - \ e) \mid k \ e & v \in V \\ \pi ::= \ b \mid \tilde{v} \mid (e \sim e) \mid \neg\pi \mid \pi \wedge \pi \mid \pi \vee \pi & \tilde{v} \in \tilde{V}, e \in E \end{array}$$

We write $vars(e)$ to mean all variables v appearing in e, and $vars(\pi)$ to mean all variables $\tilde{v}$ and v appearing in π or in e in π. We also write $atoms(\pi)$ (the set of atomic predicates) to mean all boolean variables $\tilde{v}$ and all comparisons $(e \sim e)$ appearing in π. We use the letters $\sigma, \eta, \theta, \nu$ (resp. $\tilde{\sigma}, \tilde{\eta}, \tilde{\nu}$) to mean mappings from sets of variables to $\mathbb{Z}$ (resp. $\mathbb{B}$). Given n mappings $\nu_i : V_i \to \mathbb{Z}$ such that $V_i \cap V_j = \emptyset$ for each $i, j : 1 \leq i \neq j \leq n$, and an expression $e \in \mathtt{exprsOf}(V)$, we write $val_{\nu_1,\ldots,\nu_n}(e)$ to mean the expression obtained by replacing each occurrence of a variable v appearing in some V_i by the corresponding $\nu_i(v)$. In a similar manner, we write $val_{\nu,\tilde{\nu},\ldots}(\pi)$ to mean the predicate obtained by replacing the occurrence of integer and boolean variables as stated by the mappings $\nu, \tilde{\nu}, etc.$ Given a mapping $\nu : V \to \mathbb{Z}$ and a set $subst = \{v_i \leftarrow k_i | 1 \leq i \leq n\}$ where variables $v_1, \ldots v_n$ are pairwise different, we write $\nu\,[subst]$ to mean the mapping ν' such that $\nu'(v_i) = k_i$ for each $1 \leq i \leq n$ and $\nu'(v) = \nu(v)$ otherwise. We abuse notation and write $\nu\,[\{v_i \leftarrow v'_i | 1 \leq i \leq n\}]$, for $\nu : V \to \mathbb{Z}$ where variables $v_1, \ldots v_n$ are in V and pairwise different, to mean the mapping $\nu' : (V \setminus \{v_i | 1 \leq i \leq n\}) \cup \{v'_i | 1 \leq i \leq n\} \to \mathbb{Z}$ and such that $\nu'(v') = \nu(v)$ for each $v \in \{v_i | 1 \leq i \leq n\}$ and $\nu'(v) = \nu(v)$ otherwise. We define $\tilde{\nu}\,[\{\tilde{v}_i \leftarrow b_i | 1 \leq i \leq n\}]$ and $\tilde{\nu}\,[\{\tilde{v}_i \leftarrow \tilde{v}'_i | 1 \leq i \leq n\}]$ in a similar manner.

A multiset m over a set X is a mapping $X \to \mathbb{Z}$. We write $x \in m$ to mean $m(x) \geq 1$. The size $|m|$ of a multiset m is $\sum_{x \in X} m(x)$. We sometimes view a multiset m as a sequence $x_1, x_2, \ldots, x_{|m|}$ where each element x appears $m(x)$ times. We write $x \oplus m$ to mean the multiset m' such that $m'(y)$ equals $m(y) + 1$ if $x = y$ and $m(y)$ otherwise.

4 Concurrent Programs and Counting Logic

To simplify the presentation, we assume a concurrent program (or program for short) to consist in a single non-recursive procedure manipulating integer variables. Arguments and return values are passed using shared variables. Programs where arbitrary many processes run a finite number of procedures can be encoded by having the processes choose a procedure at the beginning.

Syntax. A procedure in a program (S, L, T) is given in terms of a set T of transitions $(lc_1 \blacktriangleright lc_1' : stmt_1), (lc_2 \blacktriangleright lc_2' : stmt_2), \ldots$ operating on two finite sets of integer variables, namely a set S of shared variables (denoted $s_1, s_2, \ldots$) and a set L of local variables (denoted $l_1, l_2 \ldots$). Each transition $(lc \blacktriangleright lc' : stmt)$ involves two locations lc and lc' and a statement $stmt$. We write Loc to mean the set of all locations appearing in T. We always distinguish two locations, namely an entry location lc_{ent} and an exit location lc_{ext}. In any transition, location lc_{ext} may appear as the source location only if the destination is also lc_{ext} (i.e., a sink location). Program syntax is given in terms of pairwise different $v_1, \ldots v_n$ in $S \cup L$, $e_1, \ldots e_n$ in $\texttt{exprsOf}(S \cup L)$ and π is in $\texttt{predsOf}(\texttt{exprsOf}(S \cup L))$.

$$\begin{array}{l} \text{prog} ::= (s := (k \mid *))^* \quad \underline{proc:} \quad (l := (k \mid *))^* \quad (lc \blacktriangleright lc : stmt)^+ \\ stmt ::= spawn \mid join \mid \pi \mid v_1, \ldots, v_n := e_1, \ldots, e_n \mid stmt; stmt \end{array}$$

Semantics. Initially, a single process starts executing the procedure with both local and shared variables initialized as stated in their definitions. Executions might involve an arbitrary number of spawned processes. The execution of any process (whether initial or spawned with the statement *spawn*) starts at the entry location lc_{ent}. Any process at an exit point lc_{ext} can be eliminated by a process executing a *join* statement. An assume π statement blocks if the predicate π over local and shared variables does not evaluate to true. Each transition is executed atomically without interruption from other processes.

More formally, a *configuration* is given in terms of a pair (σ, m) where the *shared state* $\sigma : S \to \mathbb{N}$ is a mapping that associates a natural number to each variable in S. An *initial shared state* (written σ_{init}) is a mapping that complies with the initial constraints for the shared variables. The multiset m contains *process configurations*, i.e., pairs (lc, η) where the location lc belongs to Loc and the *process state* $\eta : L \to \mathbb{N}$ maps each local variable to a natural number. We also write η_{init} to mean an *initial process state*. An *initial multiset* (written m_{init}) maps all (lc, η) to 0 except for a single (lc_{ent}, η_{init}) mapped to 1. We introduce a relation $\overset{stmt}{\rightarrowtail}_P$ in order to define statements semantics (Fig. 3). We write $(\sigma, \eta, m) \overset{stmt}{\rightarrowtail}_P (\sigma', \eta', m')$, where σ, σ' are shared states, η, η' are process states, and m, m' are multisets of process configurations, in order to mean that a process at process state η when the shared state is σ and the other process configurations are represented by m, can execute the statement *stmt* and take the program to a configuration where the process is at state η', the shared

state is σ' and the configurations of the other processes are captured by m'. For instance, a process can always execute a join if there is another process at location lc_{ext} (rule $join$). A process executing a multiple assignment atomically updates shared and local variables values according to the values taken by the expressions of the assignment before the execution (rule $assign$).

$$\frac{(\sigma,\eta,m) \overset{stmt}{\rightarrowtail_P} (\sigma',\eta',m')}{(\sigma,(lc,\eta)\oplus m) \overset{(lc \blacktriangleright lc' : stmt)}{\longrightarrow_P} (\sigma',(lc',\eta')\oplus m')} : trans \qquad \frac{val_{\sigma,\eta}(\pi)}{(\sigma,\eta,m) \overset{\pi}{\rightarrowtail_P} (\sigma,\eta,m)} : assume$$

$$\frac{(\sigma,\eta,m) \overset{stmt}{\rightarrowtail_P} (\sigma',\eta',m') \quad (\sigma',\eta',m') \overset{stmt'}{\rightarrowtail_P} (\sigma'',\eta'',m'')}{(\sigma,\eta,m) \overset{stmt;stmt'}{\rightarrowtail_P} (\sigma'',\eta'',m'')} : seq \qquad \frac{m = \left((lc_{ext},\eta')\oplus m'\right)}{(\sigma,\eta,m) \overset{join}{\rightarrowtail_P} (\sigma,\eta,m')} : join$$

$$\frac{subst_A = \left\{v_i \leftarrow val_{\sigma',\eta'}(e_i) \,|\, v_i \in A\right\}}{(\sigma,\eta,m) \overset{v_1,\ldots v_n,:=e_1,\ldots e_n}{\rightarrowtail_P} (\sigma[subst_S],\eta[subst_L],m)} : assign \qquad \frac{m' = (lc_{ent},\eta_{init})\oplus m}{(\sigma,\eta,m) \overset{spawn}{\rightarrowtail_P} (\sigma,\eta,m')} : spawn$$

Fig. 3. Semantics of concurrent programs.

We write $(\sigma,m) \longrightarrow_P (\sigma',m')$ if $(\sigma,m) \overset{t}{\longrightarrow_P} (\sigma',m')$ for a transition t. A P *run* ρ is a sequence $(\sigma_0,m_0),t_1,...,t_n,(\sigma_n,m_n)$. The run is P *feasible* if $(\sigma_i,m_i) \overset{t_{i+1}}{\longrightarrow_P} (\sigma_{i+1},m_{i+1})$ for each $i \;:\; 0 \leq i < n$ and σ_0 and m_0 are initial. We say that a configuration (σ,m) is *reachable* if there is a feasible P run $(\sigma_0,m_0),t_1,...,t_n,(\sigma_n,m_n)$ s.t. $(\sigma,m)=(\sigma_n,m_n)$.

Counting Logic. We use $@Loc$ to mean the set $\{@lc \mid lc \in Loc\}$ of boolean variables. Intuitively, $@lc$ evaluates to `tt` exactly when the process evaluating it is at location lc. We associate a *counting variable* $(\pi)^{\#}$ to each predicate π in $\texttt{predsOf}(@Loc,\texttt{exprsOf}(S\cup L))$. Intuitively, in a given program configuration, the variable $(\pi)^{\#}$ counts the number of processes for which the predicate π holds. We let Ω mean the set $\left\{(\pi)^{\#} | \pi \in \texttt{predsOf}(@Loc,\texttt{exprsOf}(S\cup L))\right\}$ of counting variables. A *counting predicate* is a predicate in $\texttt{predsOf}(@Loc,\texttt{exprsOf}(S\cup\Omega))$.

Elements in $\texttt{exprsOf}(S\cup L)$ and $\texttt{predsOf}(@Loc,\texttt{exprsOf}(S\cup L))$ are evaluated wrt. a shared configuration σ and a process configuration (lc,η). For instance, $val_{\sigma,(lc,\eta)}(v)$ is $\sigma(v)$ if $v\in S$ and $\eta(v)$ if $v\in L$ and $val_{\sigma,(lc,\eta)}(@lc') = (lc=lc')$. We abuse notation and write $val_{\sigma,m}(\omega)$ to mean the evaluation of the counting predicate ω wrt. a configuration (σ,m). More precisely, $val_{\sigma,m}\left((\pi)^{\#}\right) = \sum_{(lc,\eta) \text{ s.t. } val_{\sigma,(lc,\eta)}(\pi)} m((lc,\eta))$ and $val_{\sigma,m}(v)=\sigma(v)$ for $v\in S$.

Our counting logic is quite expressive as we can capture assertion violations and deadlocks. For location lc, we let $enabled(lc)$ in $\texttt{predsOf}(\texttt{exprsOf}(S\cup L))$ define when a process can fire some transition from lc. We capture the violation of an $assert(\pi)$ at some location lc and the deadlock configurations using the following two counting predicates.

$$\omega_{assert} = (@lc \wedge \neg\pi)^{\#} \geq 1 \qquad \omega_{deadlock} = \textstyle\bigwedge_{lc\in Loc}(@lc \wedge enabled(lc))^{\#} = 0$$

5 Relating layers of abstractions

We formally describe in the following the four steps involved in our predicated constrained monotonic abstraction approach (see Fig. 2).

5.1 Predicate abstraction

Given a program $P = (S, L, T)$ and a number of predicates Π on the variables $S \cup L$, we leverage on existing techniques (such as [5]) in order to generate an abstraction in the form of a boolean program $\tilde{P} = \left(\tilde{S}, \tilde{L}, \tilde{T}\right)$ where all shared and local variables take boolean values. To achieve this, Π is partitioned into three sets Π_{shr}, Π_{loc} and Π_{mix}. Predicates in Π_{shr} only mention variables in S and those in Π_{loc} only mention variables in L. Predicates in Π_{mix} mention both shared and local variables of P. A one to one mapping associates a predicate $\texttt{origPredOf}(\tilde{v})$ in Π_{shr} (resp. $\Pi_{mix} \cup \Pi_{loc}$) to each $\tilde{v}$ in $\tilde{S}$ (resp. $\tilde{L}$).

In addition, there are as many transitions in T as in $\tilde{T}$. For each $(lc \blacktriangleright lc' : stmt)$ in T there is a corresponding $(lc \blacktriangleright lc' : \texttt{abstOf}(stmt))$ with the same source and destination locations lc, lc', but with an abstracted statement $\texttt{abstOf}(stmt)$ that may operate on the variables $\tilde{S} \cup \tilde{L}$. For instance, the statement $(count := count + 1)$ in Fig. 1 is abstracted with the multiple assignment:

$$\begin{pmatrix} wait\_leq\_count, \\ count\_leq\_wait \end{pmatrix} := \begin{pmatrix} choose\,(wait\_leq\_count, \texttt{ff})\,, \\ choose\,(\neg wait\_leq\_count \wedge count\_leq\_wait, wait\_leq\_count) \end{pmatrix} \quad (1)$$

The value of the variable *count_leq_wait* after execution of the multiple assignment 1 is `tt` if $\neg wait\_leq\_count \wedge count\_leq\_wait$ holds, `ff` if *wait_leq_count* holds, and is equal to a non deterministically chosen boolean value otherwise. In addition, abstracted statements can mention the local variables of passive processes, i.e., processes other than the one executing the transition. For this, we make use of the variables $\tilde{L}_p = \left\{\tilde{l}_p | \tilde{l} \text{ in } \tilde{L}\right\}$ where each $\tilde{l}_p$ denotes the local variable $\tilde{l}$ of passive processes. For instance, the statement $prev := val$ in Fig. 1 is abstracted with the multiple assignment 2. Here, the local variable *prev_leq_val* of each process other than the one executing the statement (written $prev\_leq\_val_p$) is separately updated. This corresponds to a broadcast where the local variables of all passive processes need to be updated.

$$\begin{pmatrix} prev\_leq\_val, \\ prev\_leq\_max, \\ prev\_leq\_val_p \end{pmatrix} := \begin{pmatrix} \texttt{tt}, \\ choose \begin{pmatrix} \neg prev\_leq\_val \\ \wedge\ prev\_leq\_max \end{pmatrix}, \begin{pmatrix} prev\_leq\_val \\ \wedge\ \neg prev\_leq\_max \end{pmatrix}, \\ choose \begin{pmatrix} \neg prev\_leq\_val \\ \wedge\ prev\_leq\_val_p \end{pmatrix}, \begin{pmatrix} prev\_leq\_val \\ \wedge\ \neg prev\_leq\_val_p \end{pmatrix} \end{pmatrix} \quad (2)$$

Syntax and semantics of boolean programs. The syntax of boolean programs is described below. Variables $\tilde{v}_1, \ldots, \tilde{v}_n$ are in $\tilde{S} \cup \tilde{L} \cup \tilde{L}_p$. Predicate π is in

$\texttt{predsOf}(\tilde{S} \cup \tilde{L})$, and predicates $\pi_1, \ldots, \pi_n$ are in $\texttt{predsOf}(\tilde{S} \cup \tilde{L} \cup \tilde{L}_p)$. We further require for the multiple assignment that if $\tilde{v}_i \in \tilde{S} \cup \tilde{L}$ then $vars(\pi_i) \subseteq \tilde{S} \cup \tilde{L}$.

$$\begin{array}{l} \text{prog} ::= (\tilde{s} := (\texttt{tt} \mid \texttt{ff} \mid *))^*\ proc :\ (\tilde{l} := (\texttt{tt} \mid \texttt{ff} \mid *))^*\ \ (lc \blacktriangleright lc : stmt)^+ \\ stmt ::= spawn \mid join \mid \pi \mid \tilde{v}_1, \ldots, \tilde{v}_n := \pi_1, \ldots, \pi_n \mid stmt; stmt \end{array}$$

Apart from the fact that all variables are now boolean, the main difference of Fig. 4 with Fig. 3 is the *assign* statement. For this, we write $(\tilde{\sigma}, \tilde{\eta}, \tilde{\eta}_p) \overset{\tilde{v}_1, \ldots \tilde{v}_n := \pi_1, \ldots \pi_n}{\longmapsto_{\tilde{P}}} (\tilde{\sigma}', \tilde{\eta}', \tilde{\eta}_p')$ to mean that $\tilde{\eta}_p'$ is obtained in the following way. First, we change the domain of $\tilde{\eta}_p$ from $\tilde{L}$ to $\tilde{L}_p$ and obtain the mapping $\tilde{\eta}_{p,1} = \tilde{\eta}_p \left[\left\{\tilde{l} \leftarrow \tilde{l}_p | \tilde{l} \in \tilde{L}\right\}\right]$, then we let $\tilde{\eta}_{p,2} = \tilde{\eta}_{p,1} \left[\left\{\tilde{v}_i \leftarrow val_{\tilde{\sigma}, \tilde{\eta}, \tilde{\eta}_{p,1}}(\pi_i) \,|\, \tilde{v}_i \in \tilde{L}_p \text{ in lhs of the assignment}\right\}\right]$. Finally, we obtain $\tilde{\eta}_p' = \tilde{\eta}_{p,2} \left[\left\{\tilde{l}_p \leftarrow \tilde{l} | \tilde{l} \in \tilde{L}\right\}\right]$. This step corresponds to a broadcast. We write $(\tilde{\sigma}, \tilde{m}) \longrightarrow_{\tilde{P}} (\tilde{\sigma}', \tilde{m}')$ if $(\tilde{\sigma}, \tilde{m}) \overset{\tilde{t}}{\longrightarrow}_{\tilde{P}} (\tilde{\sigma}', \tilde{m}')$ for some $\tilde{t}$. A $\tilde{P}$ run $\tilde{\rho}$ is a sequence $(\tilde{\sigma}_0, \tilde{m}_0), \tilde{t}_1, ..., \tilde{t}_n, (\tilde{\sigma}_n, \tilde{m}_n)$. The run is *feasible* if $(\tilde{\sigma}_i, \tilde{m}_i) \overset{\tilde{t}_{i+1}}{\longrightarrow}_{\tilde{P}} (\tilde{\sigma}_{i+1}, \tilde{m}_{i+1})$ for each $i \ :\ 0 \leq i < n$ and both $\tilde{\sigma}_0$ and $\tilde{m}_0$ are initial.

$$\frac{(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \overset{stmt}{\rightarrowtail}_{\tilde{P}} (\tilde{\sigma}', \tilde{\eta}', \tilde{m}')}{(\tilde{\sigma}, (lc, \tilde{\eta}) \oplus \tilde{m}) \overset{(lc \blacktriangleright lc' : stmt)}{\longrightarrow}_{\tilde{P}} (\tilde{\sigma}', (lc', \tilde{\eta}') \oplus \tilde{m}')} : trans \qquad \frac{val_{\tilde{\sigma}, \tilde{\eta}}(\pi)}{(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \overset{\pi}{\rightarrowtail}_{\tilde{P}} (\tilde{\sigma}, \tilde{\eta}, \tilde{m})} : assume$$

$$\frac{(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \overset{stmt}{\rightarrowtail}_{\tilde{P}} (\tilde{\sigma}', \tilde{\eta}', \tilde{m}') \text{ and } (\tilde{\sigma}', \tilde{\eta}', \tilde{m}') \overset{stmt'}{\rightarrowtail}_{\tilde{P}} (\tilde{\sigma}'', \tilde{\eta}'', \tilde{m}'')}{(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \overset{stmt; stmt'}{\rightarrowtail}_{\tilde{P}} (\tilde{\sigma}'', \tilde{\eta}'', \tilde{m}'')} : sequence$$

$$\frac{\tilde{m}' = (lc_{ent}, \tilde{\eta}_{init}) \oplus \tilde{m}}{(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \overset{spawn}{\rightarrowtail}_{\tilde{P}} (\tilde{\sigma}, \tilde{\eta}, \tilde{m}')} : spawn \qquad \frac{\tilde{m} = \left((lc_{ext}, \tilde{\eta}') \oplus \tilde{m}'\right)}{(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \overset{join}{\rightarrowtail}_{\tilde{P}} (\tilde{\sigma}, \tilde{\eta}, \tilde{m}')} : join$$

$$\frac{\begin{array}{c} \tilde{\sigma}' = \tilde{\sigma}[\left\{\tilde{v}_i \leftarrow val_{\tilde{\sigma}, \tilde{\eta}}(\pi_i) \,|\, \tilde{v}_i \in \tilde{S}\right\}] \\ \tilde{\eta}' = \tilde{\eta}[\left\{\tilde{v}_i \leftarrow val_{\tilde{\sigma}, \tilde{\eta}}(\pi_i) \,|\, \tilde{v}_i \in \tilde{L}\right\}] \\ h : \tilde{m} \to \tilde{m}' \text{ a bijection with } h((lc_p, \tilde{\eta}_p)) = (lc_p, \tilde{\eta}_p') \\ \text{for some } \tilde{\eta}' \text{ s.t. } (\tilde{\sigma}, \tilde{\eta}, \tilde{\eta}_p) \overset{\tilde{v}_1, \ldots \tilde{v}_n := \pi_1, \ldots \pi_n}{\mapsto_{\tilde{P}}} (\tilde{\sigma}', \tilde{\eta}', \tilde{\eta}_p') \end{array}}{(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \overset{\tilde{v}_1, \ldots \tilde{v}_n := \pi_1, \ldots \pi_n}{\rightarrowtail_{\tilde{P}}} (\tilde{\sigma}', \tilde{\eta}', \tilde{m}')} : assign$$

Fig. 4. Semantics of boolean concurrent programs.

Relation between P and $\tilde{P}$. Given a shared configuration $\tilde{\sigma}$, we write $\texttt{origPredOf}(\tilde{\sigma})$ to mean the predicate $\bigwedge_{\tilde{s} \in \tilde{S}} (\tilde{\sigma}(\tilde{s}) \Leftrightarrow \texttt{origPredOf}(\tilde{s}))$. In a similar manner, we write $\texttt{origPredOf}(\tilde{\eta})$ to mean the predicate $\bigwedge_{\tilde{l} \in \tilde{L}} (\tilde{\eta}(\tilde{l}) \Leftrightarrow \texttt{origPredOf}(\tilde{l}))$. Observe $vars(\texttt{origPredOf}(\tilde{\sigma})) \subseteq S$ and $vars(\texttt{origPredOf}(\tilde{\eta})) \subseteq S \cup L$. We abuse notation and write $val_\sigma(\tilde{\sigma})$ (resp. $val_\eta(\tilde{\eta})$) to mean that $val_\sigma(\texttt{origPredOf}(\tilde{\sigma}))$

(resp. $val_\eta(\texttt{origPredOf}(\tilde{\eta}))$) holds. We also write $val_{\tilde{\sigma},\tilde{\eta}}(\pi)$, for a boolean combination π of predicates in Π, to mean the predicate obtained by replacing each π' in $\Pi_{mix} \cup \Pi_{loc}$ (resp. Π_{shr}) with $\tilde{\eta}(\tilde{v})$ (resp. $\tilde{\sigma}(\tilde{v})$) where $\texttt{origPredOf}(\tilde{v}) = \pi'$. We let $val_m(\tilde{m})$ mean there is a bijection $h : |m| \rightarrow |\tilde{m}|$ s.t. we can associate to each $(lc, \eta)_i$ in m an $(lc, \tilde{\eta})_{h(i)}$ in $\tilde{m}$ such that $val_\eta(\tilde{\eta})$ for each $i : 1 \leq i \leq |m|$. To each $\tilde{P}$ configuration $(\tilde{\sigma}, \tilde{m})$ corresponds a set $\gamma((\tilde{\sigma}, \tilde{m})) = \{(\sigma, m) | val_\sigma(\tilde{\sigma}) \wedge val_m(\tilde{m})\}$, and to each $\tilde{P}$ configuration (σ, m) a singleton $\alpha((\sigma, m)) = \{(\tilde{\sigma}, \tilde{m}) | val_\sigma(\tilde{\sigma}) \wedge val_m(\tilde{m})\}$. We initialize the $\tilde{P}$ variables s.t. for each σ_{init}, m_{init} there are $\tilde{\sigma}_{init}, \tilde{m}_{init}$ s.t. $\alpha((\sigma_{init}, m_{init})) = \{(\tilde{\sigma}_{init}, \tilde{m}_{init})\}$. To each $\tilde{P}$ run $\tilde{\rho} = (\tilde{\sigma}_0, \tilde{m}_0), \tilde{t}_1, ...(\tilde{\sigma}_n, \tilde{m}_n)$ corresponds a set of P runs $\gamma(\tilde{\rho}) = \left\{(\sigma_0, m_0), t_1, ...(\sigma_n, m_n) | (\sigma_i, m_i) \in \gamma((\tilde{\sigma}, \tilde{m})), \tilde{t}_i = \texttt{abstOf}(t_i)\right\}$, and to each P run $\rho = (\sigma_0, m_0), t_1, ...t_n, (\sigma_n, m_n)$ corresponds a set of $\tilde{P}$ runs $\alpha(\rho)$ defined as $\left\{\alpha((\sigma_0, m_0)), \tilde{t}_1, ...\tilde{t}_n, \alpha((\sigma_n, m_n)) | t_i = \texttt{abstOf}(t_i)\right\}$.

Definition 1 (predicate abstraction). *Let $P = (S, L, T)$ be a program and $\tilde{P} = \left(\tilde{S}, \tilde{L}, \tilde{T}\right)$ be its abstraction wrt. Π as described in this Section. The abstraction is said to be effective and sound if $\tilde{P}$ can be effectively computed and to each feasible P run ρ corresponds a non empty set $\alpha(\rho)$ of feasible $\tilde{P}$ runs.*

5.2 Translation into an extended counter machine

Assume a program $P = (S, L, T)$, a set $\Pi \subseteq \texttt{predsOf}(\texttt{exprsOf}(S \cup L))$ of predicates and two counting predicates in $\texttt{predsOf}(@Loc, \texttt{exprsOf}(S \cup \Omega))$: an invariant ω_{inv} and a target ω_{trgt}. We write $\tilde{P} = \left(\tilde{S}, \tilde{L}, \tilde{T}\right)$ to mean the boolean abstraction of P wrt. $\Pi \cup atoms(\omega_{trgt}) \cup atoms(\omega_{inv})$. Intuitively, this step results in the state reachability problem of an extended counter machine $M_{P,\Pi,\omega_{inv},\omega_{trgt}}$ that captures reachability of $\tilde{P}$ configurations (abstracting ω_{trgt} P configurations) with $\tilde{P}$ runs that are strengthened wrt. the P counting invariant ω_{inv}.

An extended counter machine M is a tuple $(Q, C, \Delta, Q_{Init}, \Theta_{Init}, q_{trgt})$ where Q is a finite set of states, C is a finite set of counters (i.e., variables ranging over $\mathbb{N}$), Δ is a finite set of transitions, $Q_{Init} \subseteq Q$ is a set of initial states, Θ_{Init} is an initial set of counters valuations, (i.e., mappings from C to $\mathbb{N}$) and q_{trgt} is a state in Q. A transition δ in Δ is of the form $[q : op : q']$ where the operation op is either the identity operation nop, a guarded command $grd \Rightarrow cmd$, or a sequential composition of operations. We use a set A of auxiliary variables ranging over $\mathbb{N}$. These are meant to be existentially quantified when firing the transitions as explained in Fig. 5. A guard grd is a predicate in $\texttt{predsOf}(\texttt{exprsOf}(A \cup C))$ and a command cmd is a multiple assignment $c_1, \ldots, c_n := e_1, \ldots, e_n$ that involves $e_1, \ldots e_n$ in $\texttt{exprsOf}(A \cup C)$ and pairwise different $c_1, \ldots c_n$.

A *machine configuration* is a pair (q, θ) where q is a state in Q and θ is a mapping $C \rightarrow \mathbb{N}$. Semantics are given in Fig. 5. A configuration (q, θ) is *initial* if $q \in Q_{Init}$ and $\theta \in \Theta_{Init}$. An M run ρ_M is a sequence $(q_0, \theta_0); \delta_1; \ldots (q_n, \theta_n)$. It is *feasible* if (q_0, θ_0) is initial and $(q_i, \theta_i) \xrightarrow{\delta_{i+1}}_M (q_{i+1}, \theta_{i+1})$ for $i : 0 \leq i < n$. We

write $\longrightarrow_M$ to mean $\cup_{\delta\in\Delta} \overset{\delta}{\longrightarrow}_M$. The machine state reachability problem is to decide whether there is an M feasible run $(q_0,\theta_0);\delta_1;\ldots(q_n,\theta_n)$ s.t. $q_n = q_{trgt}$.

$$\frac{\delta = [q : op : q'] \text{ and } \theta \overset{op}{\rightarrowtail}_M \theta'}{(q,\theta) \overset{\delta}{\longrightarrow}_M (q',\theta')} : transition$$

$$\frac{}{\theta \overset{nop}{\rightarrowtail}_M \theta} : nop \qquad \frac{\theta \overset{op}{\rightarrowtail}_M \theta' \text{ and } \theta' \overset{op'}{\rightarrowtail}_M \theta''}{\theta \overset{op;op'}{\rightarrowtail}_M \theta''} : seq$$

$$\frac{\exists A. val_\theta(grd) \wedge \forall i : 1 \le i \le n. \theta'(c_i) = val_\theta(e_i)}{\theta \overset{grd \Rightarrow cmd}{\rightarrowtail}_M \theta'} : gcmd$$

Fig. 5. Semantics of an extended counter machine

Translation. We describe a machine $(Q, C, \Delta, Q_{Init}, \Theta_{Init}, q_{trgt})$ that captures the behaviour of the program $\tilde{P}$ and encodes reaching abstractions of P configurations satisfying ω_{trgt} in terms of a machine state reachability problem. Each state in Q is either the target state q_{trgt} or is associated to a shared configuration $\tilde{\sigma}$. We write $q_{\tilde{\sigma}}$ to make the association explicit. There is a one to one mapping that associates a process configuration $(lc,\tilde{\eta})$ to each counter $c_{(lc,\tilde{\eta})}$ in C. Transitions are generated as described in Fig. 7. We associate a program configuration $(\tilde{\sigma},\tilde{m})$ to each machine configuration $enc_M((\tilde{\sigma},\tilde{m})) = (q_{\tilde{\sigma}},\theta)$. Intuitively, states in Q encode values of the shared variables while each counter $c_{(lc,\tilde{\eta})}$ counts the number of processes at location lc and satisfying $\tilde{\eta}$. In other words $\tilde{m}((lc,\tilde{\eta})) = c_{(lc,\tilde{\eta})}$ for each $(lc,\tilde{\eta})$. The encoding of a $\tilde{P}$ run $\rho_{\tilde{P}} = (\tilde{\sigma}_0,\tilde{m}_0);\tilde{t}_1;\ldots(\tilde{\sigma}_n,\tilde{m}_n)$ is the set $\left\{(q_0,\theta_0);\delta_1\ldots;(q_n,\theta_n) | (q_i,\theta_i) = enc_M((\tilde{\sigma}_i,\tilde{m}_i)) \text{ and } \delta_i \in \Delta_{\tilde{t}_i}\right\}$. The machine encodes the behaviour of the boolean program as specified in Lem. 1.

Lemma 1 (monotonic translation). *Any $\tilde{P}$ configuration $(\tilde{\sigma},\tilde{m})$ such that $\omega_{trgt}\left[\texttt{origPredOf}(\tilde{s})/\tilde{\sigma}(\tilde{s})\right]\left[(\pi)^{\#}/\sum_{\{(lc,\tilde{\eta})|val_{(lc,\tilde{\eta})}(\pi)\}} \tilde{m}((lc,\tilde{\eta}))\right]$ holds is $\tilde{P}$ reachable iff q_{trgt} is M reachable.*

Let us write $\theta \preceq \theta'$ if $\theta(c) \le \theta'(c)$ for each $c \in C$. Observe that all machine transitions of Fig. 7 are monotonic in that if $(q_1,\theta_1) \overset{\delta}{\longrightarrow}_M (q_2,\theta_2)$ for some $\delta \in \Delta$, and if $\theta_1 \preceq \theta_3$, then there is a θ_4 such that $\theta_2 \preceq \theta_4$ and $(q_1,\theta_3) \overset{\delta}{\longrightarrow}_M (q_2,\theta_4)$. Combined with the fact that $\preceq$ is a well quasi ordering over $\mathbb{N}^{|C|}$, we get that:

Lemma 2 (monotonic decidability). *State reachability of any monotonic translation is decidable.*

However, monotonic translations correspond to coarse over-approximations that are incapable of dealing with statements of our counting logic (Sec. 4). Intuitively, bad configurations (such as those where a deadlock occurs) are no more

$$\frac{[q_{\tilde{\sigma}} : op : q_{\tilde{\sigma}'}] \in \Delta \quad strong_{\omega_{inv}}(\tilde{\sigma}) = \exists S.\texttt{origPredOf}(\tilde{\sigma}) \wedge \omega_{inv}\left[(\pi)^{\#} / \sum_{\{(lc,\tilde{\eta})|val_{(lc,\tilde{\eta})}(\pi)\}} c_{(lc,\tilde{\eta})}\right]}{[q_{\tilde{\sigma}} : strong_{\omega_{inv}}(\tilde{\sigma}); op; strong_{\omega_{inv}}(\tilde{\sigma}') : q_{\tilde{\sigma}'}] \in \Delta'} \; strengthen$$

Fig. 6. Strengthening of counter transitions given a counting invariant ω_{inv}.

upward closed wrt. $\preceq$. This loss of precision makes the verification of such properties out of the reach of techniques solely based on monotonic translations. To regain some of the lost precision, we constrain the runs using counting invariants.

Lemma 3 (strengthened soundness). *Any feasible P run ρ_P has a $\tilde{P}$ feasible run $\rho_{\tilde{P}}$ in $\alpha(\rho_P)$ with an M feasible run in $enc_M(\rho_{\tilde{P}})$, where M is any machine strengthened as described in Fig. 7 and Fig. 6.*

$$\frac{(lc \blacktriangleright lc' : stmt) \text{ and } [(\tilde{\sigma}, \tilde{\eta}) : op : (\tilde{\sigma}', \tilde{\eta}')]_{stmt}}{(q_{\tilde{\sigma}} : c_{(lc,\tilde{\eta})} \geq 1; (c_{(lc,\tilde{\eta})})^{--}; op; (c_{(lc',\tilde{\eta}')})^{++} : q_{\tilde{\sigma}'}) \in \Delta_{(lc \blacktriangleright lc' : stmt)}} : transition$$

$$\frac{(lc \blacktriangleright lc' : stmt) \text{ and } [(\tilde{\sigma}, \tilde{\eta}) : op : (\tilde{\sigma}', \tilde{\eta}')]_{stmt}}{(q_{\tilde{\sigma}} : \omega_{trgt}[\tilde{s}/\tilde{\sigma}(\tilde{s})]\left[(\pi)^{\#} / \sum_{(lc,\tilde{\eta}) \models \pi} c_{(lc,\tilde{\eta})}\right] : q_{trgt}) \in \Delta_{trgt}} : target$$

$$\frac{val_{\tilde{\sigma},\tilde{\eta}}(\pi)}{[(\tilde{\sigma}, \tilde{\eta}) : nop : (\tilde{\sigma}, \tilde{\eta})]_{\pi}} : assume \quad \frac{}{\left[(\tilde{\sigma}, \tilde{\eta}) : (c_{(lc_{ent}, \tilde{\eta}_{init})})^{++} : (\tilde{\sigma}, \tilde{\eta})\right]_{spawn}} : spawn$$

$$\frac{[(\tilde{\sigma}, \tilde{\eta}) : op : (\tilde{\sigma}', \tilde{\eta}')]_{stmt} \text{ and } [(\tilde{\sigma}', \tilde{\eta}') : op' : (\tilde{\sigma}'', \tilde{\eta}'')]_{stmt'}}{[(\tilde{\sigma}, \tilde{\eta}) : op; op' : (\tilde{\sigma}'', \tilde{\eta}'')]_{stmt;stmt'}} : sequence$$

$$\frac{}{\left[(\tilde{\sigma}, \tilde{\eta}) : c_{(lc_{ext}, \tilde{\eta}')} \geq 1; (c_{(lc_{ext}, \tilde{\eta}')})^{--} : (\tilde{\sigma}, \tilde{\eta})\right]_{join}} : join$$

$$\frac{\begin{array}{c}\tilde{\sigma}' = \tilde{\sigma}[\left\{\tilde{v}_i \leftarrow val_{\tilde{\sigma},\tilde{\eta}}(\pi_i) \,|\tilde{v}_i \in \tilde{S}\right\}] \quad \tilde{\eta}' = \tilde{\eta}[\left\{\tilde{v}_i \leftarrow val_{\tilde{\sigma},\tilde{\eta}}(\pi_i) \,|\tilde{v}_i \in \tilde{L}\right\}] \\ E = \left\{a_{(lc,\tilde{\eta}_p),(lc,\tilde{\eta}'_p)} | (\tilde{\sigma}, \tilde{\eta}, \tilde{\eta}_p) \xrightarrow{\tilde{v}_1,\ldots\tilde{v}_n,:=\pi_1,\ldots\pi_n}_{M} (\tilde{\sigma}', \tilde{\eta}', \tilde{\eta}'_p), a_{\_} \in A\right\} \\ Dom_E = \left\{(lc, \tilde{\eta}_p) | a_{(lc,\tilde{\eta}_p),(lc,\tilde{\eta}'_p)} \in E\right\} \quad Img_E = \left\{(lc, \tilde{\eta}'_p) | a_{(lc,\tilde{\eta}_p),(lc,\tilde{\eta}'_p)} \in E\right\}\end{array}}{\left[(\tilde{\sigma}, \tilde{\eta}) : \left(\begin{array}{l}\left\{c_{(d)} = \sum_{i \in Img_E} a_{d,i} | d \in Dom_E\right\} \\ \cup \left\{c_{(i)} := \sum_{d \in Dom_E} a_{d,i} | i \in Img_E\right\}\end{array}\right) : (\tilde{\sigma}', \tilde{\eta}')\right]_{\tilde{v}_1,\ldots\tilde{v}_n,:=\pi_1,\ldots\pi_n}} : assign$$

Fig. 7. Translation of the transitions of a boolean program $\left(\tilde{S}, \tilde{L}, \tilde{T}\right)$, given a counting target ω_{trgt}, to the transitions $\Delta = \cup_{t \in \tilde{T}} \Delta_t \cup \Delta_{trgt}$ of a counter machine.

The resulting machine is not monotonic in general and we can encode the state reachability of a two counter machine.

Lemma 4 (strengthened undecidability). *State reachability is in general undecidable after strengthening.*

5.3 Constrained monotonic abstraction and preorder refinement

This step addresses the state reachability problem for an extended counter machine $M = (Q, C, \Delta, Q_{Init}, \Theta_{Init}, q_{trgt})$. As stated in Lem. 4, this problem is in general undecidable for strengthened translations. The idea here [10] is to force monotonicity with respect to a well-quasi ordering $\preceq$ on the set of its configurations. This is apparent at line 7 of the classical working list algorithm Alg. 1. We start with the natural component wise preorder $\theta \preceq \theta'$ defined as $\wedge_{c \in C} \theta(c) \leq \theta'(c)$. Intuitively, $\theta \preceq \theta'$ holds if θ' can be obtained by "adding more processes to" θ. The algorithm requires that we can compute membership (line 5), upward closure (line 7), minimal elements (line 7) and entailment (lines 9, 13, 15) wrt. to preorder $\preceq$, and predecessor computations of an upward closed set (line 7).

If no run is found, then `not_reachable` is returned. Otherwise a run is obtained and simulated on M. If the run is possible, it is sent to the fourth step of our approach (described in Sect. 5.4). Otherwise, the upward closure step $\mathbf{Up}_{\preceq}((q, \theta))$ responsible for the spurious trace is identified and an interpolant I (with $vars(I) \subseteq C$) is used to refine the preorder as follows: $\preceq_{i+1} := \{(\theta, \theta') | \theta \preceq_i \theta' \wedge (\theta \models I \Leftrightarrow \theta' \models I)\}$. Although stronger, the new preorder is again a well quasi ordering and the trace is guaranteed to be eliminated in the next round. We refer the reader to [3] for more details.

Lemma 5 (CMA [3]). *All steps involved in Alg. 1 are effectively computable and each instantiation of Alg. 1 is sound and terminates given the preorder is a well quasi ordering.*

5.4 Simulation on the original concurrent program

A given run of the extended counter machine $(Q, C, \Delta, Q_{Init}, \Theta_{Init}, q_{trgt})$ is simulated by this step on the original concurrent program $P = (S, L, T)$. This is possible because to each step of the counter machine run corresponds a unique and concrete transition of P. This step is classical in counter example guided abstraction refinement approaches. In our case, we need to differentiate the variables belonging to different processes during the simulation. As usual in such frameworks, if the trace turns out to be possible then we have captured a concrete run of P that violates an assertion and we report it. Otherwise, we deduce predicates that make the run infeasible and send them to step 1 (Sect. 5.1).

Theorem 1 (predicated constrained monotonic abstraction). *Assume an effective and sound predicate abstraction. If the constrained monotonic abstraction step returns* `not_reachable`*, then no configuration satisfying* ω_{trgt} *is*

input : A machine $(Q, C, \Delta, Q_{Init}, \Theta_{Init}, q_{trgt})$ and a preorder $\preceq$
output: `not_reachable` or a *run* $(q_1, \theta_1); \delta_1; (q_2, \theta_2); \delta_2; \ldots \delta_n; (q_{trgt}, \theta)$

```
1  Working := ∪_{e∈Min_⪯(ℕ^|C|)} {((q_trgt, e), (q_trgt, e))}, Visited := {};
2  while Working ≠ {} do
3  |  ((q, θ), ρ) =pick a member from Working;
4  |  Visited ∪= {((q, θ), ρ)};
5  |  if (q, θ) ∈ Q_Init × Θ_Init then return ρ;
6  |  foreach δ ∈ Δ do
7  |  |  pre = Min_⪯(Pre_δ(Up_⪯((q, θ))));
8  |  |  foreach (q', θ') ∈ pre do
9  |  |  |  if θ'' ⪯ θ' for some ((q', θ''), _) in Working ∪ Visited then
10 |  |  |  |  continue;
11 |  |  |  else
12 |  |  |  |  foreach ((q', θ''), _) ∈ Working do
13 |  |  |  |  |  if θ' ⪯ θ'' then Working = Working \ {((q', θ'), _)};
14 |  |  |  |  foreach ((q', θ''), _) ∈ Visited do
15 |  |  |  |  |  if θ' ⪯ θ'' then Visited = Visited \ {((q', θ'), _)};
16 |  |  |  |  Working ∪= {((q', θ'), (q', θ'); δ; ρ)}
17 return not_reachable;
```

Algorithm 1: Constrained monotonic abstraction

reachable in P. If a P run is returned by the simulation step, then it reaches a configuration where ω_{trgt} holds. Every iteration of the outer loop terminates given the inner loop terminates. Every iteration of the inner loop terminates.

Notice that there is no general guaranty that we establish or refute the safety property. For instance, it may be the case that one of the loops does not terminate (although each one of their iterations does) or that we need to add predicates relating local variables of two different processes (something the predicate abstraction framework we use cannot express).

6 Experimental results

We report on experiments with our prototype PACMAN(for predicated constrained monotonic abstraction). We have conducted our experiments on an Intel Xeon 2.67GHz processor with 8GB of RAM. To the best of our knowledge, the reported examples which need refinement of monotonic abstraction's preorder cannot be verified by previous techniques; either because the examples require stronger orderings than the usual preorder, or because they involve counting target predicates that are not expressed in terms of violations of program assertions.

All predicate abstraction predicates and counting invariants have been derived automatically. For the counting invariants, we implemented a thread modular analysis operating on the polyhedra numerical domain. This took less than 11 seconds for all the examples we report here. For each example, we report on the

Table 1. Checking assertion violation with PACMAN

example	P	ECM	outer loop		inner loop		results	
			num.	preds.	num.	preds.	time(s)	output
max	5:2:8	18:16:104	4	5	6	2	192	correct
max-bug	5:2:8	18:8:55	3	4	5	2	106	trace
max-nobar	5:2:8	18:4:51	3	3	3	0	24	trace
readers-writers	3:3:10	9:64:121	5	6	5	0	38	correct
readers-writers-bug	3:3:10	9:7:77	3	3	3	0	11	trace
parent-child	2:3:10	9:16:48	3	4	5	2	73	correct
parent-child -nobar	2:3:10	9:1:16	2	1	2	0	3	trace
simp-bar	5:2:9	8:16:123	3	3	5	2	93	correct
simp-nobar	5:2:9	8:7:67	3	2	3	0	13	trace
dynamic-barrier	5:2:8	8:8:44	3	3	3	0	8	correct
dynamic-barrier-bug	5:2:8	8:1:14	2	1	2	0	3	trace
as-many	3:2:6	8:4:33	3	2	6	3	62	correct
as-many-bug	3:2:6	8:1:9	2	1	2	0	2	trace

number of transitions and variables both in P and in the resulting counter machine. We also state the number of refinement steps and predicates automatically obtained in both refinement loops. We made use of several optimizations. For instance, we discarded boolean mappings corresponding to unsatisfiable combinations of predicates, we used automatically generated invariants (such as $(wait \leq count) \wedge (wait \geq 0)$ for the max example in Fig.1) to filter the state space. Such heuristics dramatically helps our state space exploration algorithms.

We report on experiments checking assertion violations in Tab.1 and deadlock freedom in Tab.2. For both cases we consider correct and buggy (by removing the barriers for instance) programs. PACMAN establishes correctness and exhibits faluty runs as expected. The tuples under the P column respectively refer to the number of variables, procedures and transitions in the origirnal program. The tuples under the ECM column refer to the number of counters, states and transitions in the extended counter machine.

Table 2. Checking deadlock with PACMAN

example	P	ECM	outer loop		inner loop		results	
			num.	preds.	num.	preds.	time(s)	output
bar-bug-no.1	4:2:7	7:16:66	4	4	6	2	27	trace
bar-bug-no.2	4:3:8	9:16:95	4	3	4	0	33	trace
bar-bug-no.3	3:2:6	6:16:78	5	4	6	1	21	trace
correct-bar	4:2:7	7:16:62	4	4	6	2	18	correct
ddlck bar-loop	4:2:10	8:8:63	3	2	3	0	16	trace
no-ddlck bar-loop	4:2:9	7:16:78	4	3	4	0	19	correct

7 Conclusions and Future Work

We have presented a technique, predicated constrained monotonic abstraction, for the automated verification of concurrent programs whose correctness depends on synchronization between arbitrary many processes, for example by means of barriers implemented using integer counters and tests. We have introduced a new logic and an iterative method based on combination of predicate, counter and monotonic abstraction. Our prototype implementation gave encouraging results and managed to automatically establish or refute program assertions deadlock freedom. To the best of our knowledge, this is beyond the capabilities of current automatic verification techniques. Our current priority is to improve scalability by leveraging on techniques such as cartesian and lazy abstraction, partial order reduction, or combining forward and backward explorations. We are also aim to generalize to richer variable types.

References

1. P. Abdulla, F. Haziza, and L. Holk. All for the price of few. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *Lecture Notes in Computer Science*, pages 476–495. Springer Berlin Heidelberg, 2013.
2. P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160:109–127, 2000.
3. P. A. Abdulla, Y.-F. Chen, G. Delzanno, F. Haziza, C.-D. Hong, and A. Rezine. Constrained monotonic abstraction: A cegar for parameterized verification. In *Proc. CONCUR 2010,* 21^{th} *Int. Conf. on Concurrency Theory*, pages 86–101, 2010.
4. E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *Proc. VMCAI '06,* 7^{th} *Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 126–141, 2006.
5. A. F. Donaldson, A. Kaiser, D. Kroening, M. Tautschnig, and T. Wahl. Counterexample-guided abstraction refinement for symmetric concurrent programs. *Formal Methods in System Design*, 41(1):25–44, 2012.
6. A. Downey. *The Little Book of SEMAPHORES (2nd Edition): The Ins and Outs of Concurrency Control and Common Mistakes.* Createspace Independent Pub, 2009.
7. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.
8. A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *Proceedings of CAV*, volume 6174 of *LNCS*, pages 654–659. Springer, 2010.
9. 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.
10. A. Rezine. *Parameterized Systems: Generalizing and Simplifying Automatic Verification.* PhD thesis, Uppsala University, 2008.

A Appendix

In this section the examples of Sec.6 are demonstrated. For simplicity the property which is going to be checked in the input program is reformulated as a statement that goes to lc_{err} which denotes the error location.

A.1 Readers and Writers

int $readcount := 0$
bool $lock := \texttt{tt},\ writing := \texttt{ff}$

$\underline{main:}$
$lc_{ent} \blacktriangleright lc_{ent} :\ spawn(writer)$
$lc_{ent} \blacktriangleright lc_{ent} :\ readcount = 0 \wedge\ lock;\ spawn(reader);\ readcount := readcount + 1$
$lc_{ent} \blacktriangleright lc_{ent} :\ readcount! = 0; spawn(reader); readcount := readcount + 1$

$\underline{reader:}$
$lc_{ent} \blacktriangleright lc_{err} :\ writing$
$lc_{ent} \blacktriangleright lc_{ext} :\ readcount = 1; readcount := readcount - 1; lock := \texttt{tt}$
$lc_{ent} \blacktriangleright lc_{ext} :\ readcount! = 1; readcount := readcount - 1$

$\underline{writer:}$
$lc_{ent} \blacktriangleright lc_1 :\ lock; lock := \texttt{ff}$
$lc_1 \blacktriangleright lc_2 :\ writing := \texttt{tt}$
$lc_2 \blacktriangleright lc_3 :\ writing := \texttt{ff}$
$lc_3 \blacktriangleright lc_{ext} :\ lock := \texttt{tt}$

Fig. 8. The readers and writers example.

The readers and writers problem is a classical problem. In this problem there is a resource which is shared between several processes. There are two type of processes, one that only read from the resource *reader* and one that read and write to it *writer*. At each time there can either exist several *reader*s or only one *writer*. *reader*s and *writer*s can not exist at the same time.

In Fig.8 a solution to the readers and writers problem with preference to readers is shown. In this approach *reader*s wait until there is no *writer* in the critical section and then get the lock that protects that section. We simulate a lock with a boolean variable *lock*. Considering the fact that in our model the transitions are atomic, such simulation is sound. When a *writer* wants to access the critical section, it first waits for the *lock* and then gets it (buy setting it to `ff`). Before starting writing, a *writer* sets a flag *writing* that we check later on in a *reader* process. At the end a *writer* unsets *writing* and frees *lock*.

An arbitrary number of *reader* processes can also be spawned. The number of *reader*s is being kept track of by the variable *readcount*. When the first *reader*

is going to be spawned (i.e. $readcount = 0$) flag *lock* must hold. *readcount* is incremented after spawning each *reader*. Whenever a *reader* starts execution, it checks flag *writing* and goes to error if it is set, because it shows that at the same time a *writer* is writing to the shared resource. When a *reader* wants to exit, it decrements the *readcount*. The last *reader* frees the *lock*.

In this example we need a counting invariant to capture the relation between number of *reader*s, i.e. *readcount* and the number of processes in different locations of process *reader*.

A.2 Parent and Child

int $i := 0$
bool $allocated := \texttt{ff}$

main :
$lc_{ent} \blacktriangleright lc_{ent}$: $spawn(parent); i := i + 1$
$lc_{ent} \blacktriangleright lc_{ent}$: $join(parent); i := i - 1$

parent :
$lc_{ent} \blacktriangleright lc_1$: $allocated := \texttt{tt}$
$lc_1 \blacktriangleright lc_2$: $spawn(child)$
$lc_2 \blacktriangleright lc_3$: $join(child)$
$lc_3 \blacktriangleright lc_{ext}$: $i = 1; allocated := \texttt{ff}$
$lc_1 \blacktriangleright lc_3$: $\texttt{tt}$

child :
$lc_{ent} \blacktriangleright lc_{ext}$: $allocated$
$lc_{ent} \blacktriangleright lc_{err}$: $\neg allocated$

Fig. 9. The Parent and Child example.

In the example of Fig.9 a sample nested spawn/join is demonstrated. In this example two types of processes exist. One is *parent* which is spawned by *main* and the other one is called *child* which is spawned by *parent*. The shared variable i is initially 0 and is incremented and decremented respectively when a *parent* process is spawned and joined. A *parent* process first sets the shared flag *allocated* and then either spawns and joins a *child* process or just moves from lc_1 to lc_3 without doing anything. The *parent* that sees $i = 1$ unsets the flag *allocated*. A child process goes to error if *allocated* is not set. This example is error free because one can see that *allocated* is unset when only one *parent* exists and that *parent* has already joined its *child* or did not spawn any *child*, i.e. no *child* exists. Such relation between number of *child* and *parent* processes as well as variable i can only be captured by appropriate counting invariants and predicate abstraction is incapable of that.

A.3 Simple Barrier

int $wait := 0, count := 0$
bool $enough :=$ `ff`, $flag := *, barrierOpen :=$ `ff`

$\underline{main:}$
$lc_{ent} \blacktriangleright lc_1 : \neg enough; spawn(proc); count := count + 1$
$lc_1 \blacktriangleright lc_{ent} : enough :=$ `ff`
$lc_1 \blacktriangleright lc_{ent} : enough :=$ `tt`

$\underline{proc:}$
$lc_{ent} \blacktriangleright lc_1 : flag :=$ `tt`
$lc_1 \blacktriangleright lc_2 : flag :=$ `ff`
$lc_2 \blacktriangleright lc_3 : wait := wait + 1$
$lc_3 \blacktriangleright lc_4 : (enough \wedge wait = count); barrierOpen :=$ `tt` $: wait := wait - 1$
$lc_3 \blacktriangleright lc_4 : barrierOpen; wait := wait - 1$
$lc_4 \blacktriangleright lc_{err} : flag$

Fig. 10. Simple Barrier example.

In the example of Fig.10 a simple application of a barrier is shown. *main* process spawns an arbitrary number of *proc*s and increments a shared variable *count* that is initially zero and counts the number of *proc*s in the program before shared flag *enough* is set. Each *proc* first sets and then unsets shared flag $flag$. The statements in lc_2 to lc_4 simulate a barrier. Each *proc* first increments a shared variable *wait* which is initially zero. Then the first *proc* that finds out that the condition $(enough \wedge wait = count)$ holds, sets a shared flag *barrierOpen* and goes to lc_4. Other *proc*s that want to traverse the barrier can the transition $lc_3 \blacktriangleright lc_4 : barrierOpen$. After the barrier a *proc* goes to error if $flag$ is unset.One can see that the error state is not reachable in this program because all *proc*s have to unset $flag$ before any of them can traverse the barrier. To prove that this example is error free, it must be shown that the barrier implementation does not let any process be in locations lc_{ent}, lc_1 or lc_2 where there are processes after barrier, i.e. in locations lc_4 and lc_{err}. Proving such property requires the relation between number of processes in program locations and variables *wait* and *count* be kept. This is possible when we use counting invariants as introduced in this paper.

A.4 Dynamic Barrier

In a dynamic barrier the number of processes that have to wait at a barrier can change. The way we implemented barriers in this paper makes it easy to capture characteristics of such barriers. In the example of Fig.11 the variables

$$
\begin{array}{l}
\textbf{int}\ \ N := *, wait := *, count := *, i := 0 \\
\textbf{bool}\ \ done := \texttt{ff} \\
\\
\underline{main:} \\
\quad lc_{ent} \blacktriangleright lc_1 : \quad count, wait := N, 0 \\
\quad\ lc_1 \blacktriangleright lc_1 : \quad i! = N, spawn(proc); i := i + 1 \\
\quad\ lc_2 \blacktriangleright lc_3 : \quad i = N \wedge wait = count \\
\quad\ lc_3 \blacktriangleright lc_3 : \quad join(proc); i := i - 1 \\
\quad\ lc_3 \blacktriangleright lc_4 : \quad i = 0; done := \texttt{tt} \\
\\
\underline{proc:} \\
\quad lc_{ent} \blacktriangleright lc_{ext} : \quad count := count - 1 \\
\quad lc_{ent} \blacktriangleright lc_{err} : \quad done
\end{array}
$$

Fig. 11. dynamic barrier

corresponding to barrier i.e. *count* and *wait* are respectively set to N and 0 in the *main*'s first statement. Then *proc*s are spawned as long as the counter i is not equal to N which denotes the total number of *proc*s in the system. Each created *proc* decrements *count* and by doing so it decrements the number of processes that have to wait at the barrier. In this example the barrier is in lc_2 of *main* and can be traversed as usual when $wait = count$ holds and no more *proc* is going to be spawned, i.e. $i = N$. Then *main* can non-deterministically join a *proc* or set flag *done* if no more *proc* exists.

A.5 As Many

$$
\begin{array}{l}
\textbf{int}\ \ count1 := 0, count2 := 0 \\
\textbf{bool}\ \ enough := \texttt{ff} \\
\\
\underline{main:} \\
\quad lc_{ent} \blacktriangleright\ lc_1\ : \quad spawn(proc1); count1 := count1 + 1 \\
\quad\ lc_1 \blacktriangleright lc_{ent} : \quad spawn(proc2); count2 := count2 + 1 \\
\quad lc_{ent} \blacktriangleright\ lc_2\ : \quad enough := \texttt{tt} \\
\\
\underline{proc1:} \\
\quad lc_{ent} \blacktriangleright\ lc_1\ : \quad enough \\
\quad\ lc_1 \blacktriangleright lc_{err} : \quad count1 \neq count2 \\
\\
\underline{proc2:} \\
\quad lc_{ent} \blacktriangleright lc_1 : \quad enough
\end{array}
$$

Fig. 12. As Many

In the example of Fig.12 process *main* spawns as many processes *proc*1 as *proc*2 and it increments their corresponding counters *count*1 and *count*2 accordingly. At some point *main* sets flag *enough* and does not spawn any other processes. Processes in *proc*1 and *proc*2 start execution after *enough* is set. A process in *proc*1 goes to error location if $count1 \neq count2$. One can see that error is not reachable because the numbers of processes in the two groups are the same and respective counter variables are initially zero and are incremented with each spawn to represent the number of processes. To verify this example obviously the relation between *count*1, *count*2 and number of processes in different locations of *proc*1 and *proc*2 must be captured.

A.6 Barriers causing deadlock

int $wait := 0, count := 0, open := 0$
bool $proceed :=$ `ff`

main :
$lc_{ent} \blacktriangleright lc_{ent}$: $spawn(proc); count := count + 1$
$lc_{ent} \blacktriangleright lc_1$: $proceed :=$ `tt`

proc :
$lc_{ent} \blacktriangleright lc_1$: $wait := wait + 1$
$lc_1 \blacktriangleright lc_2$: $proceed \wedge wait = count; open := open + 1$
$lc_1 \blacktriangleright lc_2$: $proceed \wedge wait \neq count$
$lc_2 \blacktriangleright lc_3$: $open > 0; open := open - 1$
$lc_2 \blacktriangleright lc_{err}$: $open = 0 \wedge (proc@lc_{ent})^{\#} = 0 \wedge (proc@lc_1)^{\#} = 0$

Fig. 13. Buggy Barrier No.1

In Fig.13 a buggy implementation of barrier is demonstrated. This example is based on an example in [6]. The barrier implementation in the book is based on semaphores and in our example the shared variable *open* which is initialized to zero plays the role of a semaphore. A buggy barrier is implemented in program locations lc_{ent} to lc_3. First process *main* spawns a number of process *proc*, increments the shared variable *count* which is supposed to count the number of *proc*s and at the end sets flag *proceed*. A *proc* increments shared variable *wait* which is aimed to count the number of *proc*s accumulated at the barrier. *proc*s must wait for the flag *proceed* to be set before they can proceed to lc_2. Each *proc* that finds out that condition $proceed \wedge wait = count$ holds increments *open*. This lets another process which is waiting at lc_2 to take the transition $lc_2 \blacktriangleright lc_3$, i.e. traverse the barrier. A deadlock situation is possible to happen in this implementation and that is when one or more processes are waiting for the condition $open > 0$ to hold, but there is no process left at lc_{ent} or lc_1 of *process* which may eventually increment *open*. In this case a *process* goes to error state.

$$
\begin{array}{l}
\textbf{int}\ \ wait := 0, count := 0 \\
\textbf{bool}\ \ proceed := \texttt{ff} \\
\\
\underline{main:} \\
\quad lc_{ent} \blacktriangleright lc_{ent} : \ \ spawn(proc1); count := count + 1 \\
\quad lc_{ent} \blacktriangleright lc_{ent} : \ \ spawn(proc2) \\
\quad lc_{ent} \blacktriangleright lc_{ext} : \ \ proceed := \texttt{tt} \\
\\
\underline{proc1:} \\
\quad lc_{ent} \blacktriangleright lc_1 \ : \ \ wait := wait + 1 \\
\quad lc_1 \blacktriangleright lc_2 \ : \ \ proceed \wedge wait = count \\
\quad lc_1 \blacktriangleright lc_{err} : \ \ proceed \wedge wait \neq count \wedge (proc1@lc_{ent})^{\#} = 0 \\
\underline{proc2:} \\
\quad lc_{ent} \blacktriangleright lc_1 : \ \ wait > 0; wait := wait - 1
\end{array}
$$

Fig. 14. Buggy Barrier No.2

In Fig.14 another buggy implementation of a barrier is demonstrated which makes deadlock possible. Process *main* non-deterministically either spawns a *proc*1 and increments *count* or spawns a *proc*2 or sets flag *proceed*. *proc*1 contains a barrier. Each process in *proc*1 increments *wait* and then waits at lc_1 for the barrier condition to hold. A *proc*2 decrements *wait* if $wait > 0$. A deadlock happens when at least a *proc*2 decrements *wait* which causes the condition in $lc_1 \blacktriangleright lc_2$ of *proc*1 to never hold. We check a deadlock situation in $lc_1 \blacktriangleright lc_{err}$ of *proc*1 which is equivalent to the situation where $proceed \wedge wait \neq count$ does not hold but there exists no process in lc_{ent} of *proc*1 that can increment *wait*.

$$
\begin{array}{l}
\textbf{int}\ \ wait := 0, count := 0 \\
\textbf{bool}\ \ proceed := \texttt{ff} \\
\\
\underline{main:} \\
\quad lc_{ent} \blacktriangleright lc_{ent} : \ \ spawn(proc); count := count + 1 \\
\quad lc_{ent} \blacktriangleright lc_1 \ : \ \ proceed := \texttt{tt} \\
\\
\underline{proc:} \\
\quad lc_{ent} \blacktriangleright lc_1 \ : \ \ wait := wait + 1 \\
\quad lc_{ent} \blacktriangleright lc_1 \ : \ \ wait > 0; wait := wait - 1 \\
\quad lc_1 \blacktriangleright lc_2 \ : \ \ proceed \wedge wait = count \\
\quad lc_1 \blacktriangleright lc_{err} : \ \ proceed \wedge wait \neq count \wedge (proc@lc_{ent})^{\#} = 0
\end{array}
$$

Fig. 15. Buggy Barrier No.3

The buggy implementation of a barrier in Fig.15 is similar to Fig.14, just that this time the *proc* itself may decrement the *wait* and thus make the barrier

condition $proceed \wedge wait = count$ never hold. A deadlock situation is detected similar to the Fig.14.

int $wait := 0, count := 0, open := 0$
bool $proceed :=$ `ff`

$\underline{main:}$
$lc_{ent} \blacktriangleright lc_{ent} : \quad spawn(proc); count := count + 1$
$lc_{ent} \blacktriangleright lc_1 : \quad proceed :=$ `tt`

$\underline{proc:}$
$lc_{ent} \blacktriangleright lc_1 : \quad wait := wait + 1$
$lc_1 \blacktriangleright lc_2 : \quad proceed \wedge wait = count; open := open + 1$
$lc_1 \blacktriangleright lc_2 : \quad proceed \wedge wait \neq count$
$lc_2 \blacktriangleright lc_3 : \quad open >= 1$
$lc_3 \blacktriangleright lc_4 : \quad wait := wait - 1;$
$lc_4 \blacktriangleright lc_{err} : \quad wait = 0 \wedge open = 0$
$lc_4 \blacktriangleright lc_{ent} : \quad wait = 0 \wedge open >= 1; open := open - 1$
$lc_4 \blacktriangleright lc_{ent} : \quad wait \neq 0$

Fig. 16. Buggy Barrier in Loop

The example in Fig.16 is based on an example in [6]. It demonstrates a buggy implementation of a reusable barrier. Reusable barriers are needed when a barrier is inside a loop. In Fig.16 the loop is formed by backward edges from lc_3 to lc_{ent}. Process *main* spawns *proc* and increments *count* accordingly. Program locations lc_{ent} to lc_3 in *proc* correspond the barrier implementation and are similar to example in Fig.13 and the other transitions make the barrier ready to be reused in the next loop iteration. The example is buggy first because deadlock is possible and second because a processes can continue to next loop iteration while others are still in previous iterations. Deadlock will happen when processes are not able to proceed from lc_4 because $wait = 0$ but $open = 0$, thus they can never take any of the $lc_4 \blacktriangleright lc_{ent}$ edges. For detecting such a deadlock scenario it is essential to capture the relation between shared variables *count* and *wait* with number of *proc*s in different locations.