Abstracting and Counting Synchronizing Processes (extended abstract)

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. Automatically checking such properties for these programs is beyond the capabilities of current verification techniques. For this purpose, we describe an original logic that mixes two sorts of variables: those shared and manipulated by the concurrent processes, and ghost variables refering to the number of processes satisfying predicates on shared and local program variables. We then combine existing works on counter, predicate, and constrained monotonic abstraction and nest two cooperating counter example based refinement loops for establishing correctness (safety 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 synchronizing processes.

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 by programs 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

* In part supported by the 12.04 CENIIT project.

violations, and global properties such as deadlocks or relations between the numbers of of processes (e.g., the total number of spawner processes is smaller or equal to the number of spawned processes).

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 already tricky for concurrent boolean programs with barriers. It is now folklore that concurrent boolean programs can be encoded using counter machines where counters track the number of processes at each program location. In case the concurent processes can only read, test and write shared boolean variables, or spawn and join other processes, the obtained counter machine is essentially a Vector Addition System (VAS) for which state reachability is decidable [3, 13]. For instance, works such as [6, 8, 9] build on this idea. Such translations cannot faithfully capture behaviours enforced by the barriers, e.g., there is no process still in the reading phase when some process crossed the barrier to the writing phase. The reason is that VASs are inherently monotonic (more processes can do more things). However, a counter machine transition that models a barrier will need to test that all processes are finished with the current phase and are waiting to cross the barrier. In other words, that the number of processes not waiting for the barrier is zero. This makes it possible to encode counter machines for which reachability is undecidable.

To make the problem more difficult, barriers may be implicitely implemented using integer program variables that count the number of processes at certain locations. Still, program correctness might depend on the fact that these program variables do implement a barrier. Existing techniques, such as symmetric predicate abstraction [8, 9], generate (broadcast) concurrent boolean programs for integer manipulating concurrent programs. The obtained transition systems are monotonic and cannot exclude behaviors forbidden by the implicit barriers. In this work, we build on such methods and strengthen the obtained transition systems using automatically generated invariants in order to obtain counter machines that over-approximate the concurrent program behavior and still faithfully capture the barriers semantics. We then build on our work on constrained monotonic abstraction [4] in order to decide state reahability by automatically generating and refining monotonic over-approximations for such systems.

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

1. We define a *counting logic* that allows us 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 [8, 9]. We encode resulting boolean programs in terms of a counter machine where reachability of the concurrent program configurations satisfying a *counting property* from our logic is captured as a reachability problem for a target state of the counter machine.
3. We explain how to strengthen the counter machine using *counting invariants*, i.e. properties from our logic that hold on all runs. We generate these invariants using classical thread modular analysis techniques [14].
4. We leverage on existing constrained monotonic abstraction techniques [17, 4] 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 automatically establishs or refutes counting properties such as deadlock freedom and assertions.

Related work. Several works consider automatic parameterized verification for concurrent programs. The works in [15, 1] automatically check for cutoff conditions. Except for checking larger instances, it is unclear how to refine entailed abstractions. Similar to [2], we combine auxiliary invariants obtained on certain variables in order to strengthen a reachability analysis. In [12], the authors propose an approach to synthetise counters in order to automatically build correctness proofs from program traces. The approach repeatedly builds safe counting automata and tries to establish that their language includes traces of a program given as a monotonic control flow net. In order to be precise, we need to over-approximate our concurrent programs with non-monotonic transition systems. In [6], the authors present a highly optimized coverability checking approach for VASs with broadcasts. We need more than coverability of monotonic systems. In [16], the authors adopt symbolic representations that can track inter-thread predicates. This yields a non monotonic system and the authors force monotonicity as in [17, 4]. They however do not explain how to refine the obtained decidable monotonic abstraction for an undecidable problem. In [5], the authors prove termination for depth-bounded systems by instrumenting a given over-approximation with counters and sending the numerical abstraction to existing termination provers. We automatically generate the abstractions on which we establish safety properties. In addition, and as stated earlier, over-approximating the concurrent programs we target with (monotonic) well structured transition systems would result in spurious runs. The works that seem most closely related are [4, 10]. We introduced (constrained) monotonic abstraction in [17, 4]. Monotonic abstraction was not combined with predicate abstraction, nor did it explicitly target counting properties or dynamic barrier based synchronization. In [10, 9], the authors propose a predicate abstraction framework for concurrent multithreaded programs. As explained earlier such abstractions cannot exclude runs forbidden by synchronization mechanisms such as barriers. In our work, we build on [10, 9] in order to handle shared and local integer variables.

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 loops in Sec. 5 and report on our experimental results in Sec. 6. We finally conclude in Sec. 7. Proofs and examples are available in the Appendix.

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 non-deterministic values. 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* 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). Observe that *prev* is essentially a ghost variable we add to check that *max* is indeed larger than any initial value of the local, and possibly modified, *val*. Violation of this assertion can be captured with the *counting predicate* (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. Observe that we could have used an error state to capture assertion violations. However, our counting logic (see Sec. 4) also allows us to express global properties (such as that there are more processes with $flag = \texttt{tt}$ than those with $flag = \texttt{ff}$). Reachability of such global configurations is easier to express with counting properties that anyhow can capture assertion violations.

The assertion $(proc@lc_5 \wedge \neg(prev \leq max))^{\#} \geq 1$ is never violated when 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 variables and the number of *proc*s that may participate in the run. 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 $prev := val$ (possibly violating the assertion) although one of them is at $proc@lc_5$.

Our nested CEGAR, 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 on the max example of Fig. 1.

From concurrent programs to boolean concurrent programs. We build on recent predicate abstraction techniques for concurrent programs [10]. Such techniques

```
int   max, prev, wait, count := *, *, 0, 0
bool  proceed := ff
main :
   t1 : lc_ent ▶ lc_ent :  count := count + 1;
                           spawn(proc)
   t2 : lc_ent ▶  lc1   :  proceed := tt
   ...

proc :
   int  val := *
   t3 : lc_ent ▶ lc1 :  prev := val
   t4 :  lc1   ▶ lc2 :  max ≥ val
   t5 :  lc1   ▶ lc2 :  max < val; max := val
   t5 :  lc2   ▶ lc3 :  val := *
   t6 :  lc3   ▶ lc4 :  wait := wait + 1
   t7 :  lc4   ▶ lc5 :  proceed ∧ (wait = count)
   t8 :  lc5   ▶ ...
```

$$(3,7,0,0,\texttt{ff})\,\{(main@lc_{ent})\}$$
$$\downarrow t_1$$
$$(3,7,0,\underline{1},\texttt{ff})\,\{(main@lc_{ent})(proc@lc_{ent},9)\}$$
$$\downarrow t_1$$
$$(3,7,0,\underline{2},\texttt{ff})\,\left\{(main@lc_{ent})\underline{(proc@lc_{ent},9)^2}\right\}$$
$$\downarrow t_3$$
$$(3,\underline{9},0,2,\texttt{ff})\,\left\{\underline{(main@lc_1)}(proc@lc_{ent},9)(proc@lc_1,9)\right\}$$
$$\downarrow t_2$$
$$(3,9,0,2,\underline{\texttt{tt}})\,\left\{\underline{(main@lc_1)}(proc@lc_{ent},9)(proc@lc_1,9)\right\}$$
$$\downarrow \cdots$$
$$\cdots$$
$$\downarrow t_7$$
$$(9,9,2,2,\underline{\texttt{tt}})\,\left\{\underline{(main@lc_1)}(proc@lc_5,9)^2\right\}$$
$$\downarrow \cdots$$
$$\cdots$$

Fig. 1. The max example (left) and a possible run (right). The run starts with the *main* process being at location lc_{ent} where $(max, prev, wait, count, proceed) = (3, 7, 0, 0, \texttt{ff})$.

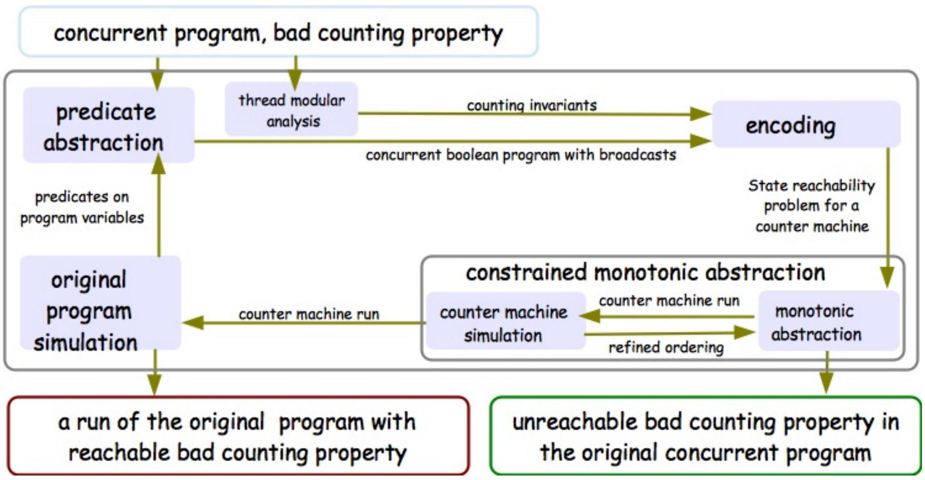

Fig. 2. **P**redicat**e**d **C**onstrained **M**onotonic **A**bstractio**n**

would initially 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 booleans. Hence, one would need a finite number of counters in order to faithfully capture the behavior of the abstracted program using counter abstraction.

From concurrent boolean programs to counter machines. Given a concurrent boolean program, we generate a monotonic counter machine for which state reachability is equivalent to the violation of the assertion by the boolen program. Each counter in the machine counts the number of processes at some location with a given valuation of the local variables. One state in the counter machine represents reaching a configuration violating the assertion. State reachability is here decidable [3, 13]. Such a machine cannot relate the number of processes in certain locations (e.g., the number of spawned processes *proc* so far) to the shared predicates that hold at a machine state (e.g., that $count = wait$). For this reason, we make use of the auxiliary invariants [2]:

$$count = \sum_{lc \in proc@Loc} (lc)^{\#} \qquad wait = \sum_{i \geq 3} (proc@lc_i)^{\#}$$

We automatically generate such invariants using a simple thread modular analysis [14] that tracks the number of processes at each location. We then strengthen the counter machine using such invariants. This results in a more precise machine for which state reachability is undecidable in general.

Constrained monotonic abstraction. We monotonically abstract the resulting counter machine in order to answer the state reachability problem. Spurious runs are now possible. Indeed, forcing monotonicity amounts to removing [17, 4] processes violating the constraint imposed by the barrier in Fig.1. Suppose now that two processes are spawned and *proceed* is set to `tt`. A first process gets to lc_3 and waits for the second process that moves to lc_1. Removing the second process (because it violates the barrier constraint) opens the barrier for the first process waiting at lc_3. 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 (i.e., the constrained monotonic abstraction loop) can automatically add more elaborate refinements such as comparing the number of processes at different locations. Unreachability of the control location establishes safety of the concurrent program.

Trace Simulation. Counter examples obtained in the counter machine correspond to feasible runs as far as the concurrent boolean program is concerned. Such runs 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 use $\mathbb{N}$ and $\mathbb{Z}$ to mean the sets of natural and integer numbers respectively. We let k denote a constant in $\mathbb{Z}$. Unless otherwise stated, we use lower case letters such as v, s, l to mean integer variables and $\tilde{v}, \tilde{s}, \tilde{l}$ to mean boolean variables with values in $\mathbb{B}$. We use upper case letters such as V, S, L (resp. $\tilde{V}, \tilde{S}$ and $\tilde{L}$) to mean sets of integer (resp. boolean) variables. We let $\sim$ be an element in $\{<, \leq, =, \geq, >\}$. An arithmetic expression e (resp. boolean expression π) belonging to the set $\texttt{exprs}(V)$ (resp. $\texttt{preds}(\tilde{V}, E)$) of arithmetic expressions (resp. boolean predicates) over integer variables V (resp. boolean variables $\tilde{V}$ and arithmetic expressions E) is defined as follows.

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

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 comparisons $(e \sim e)$ appearing in π. We use greek lower case letters such as σ, η, ν (resp. $\tilde{\sigma}, \tilde{\eta}, \tilde{\nu}$) to mean mappings from 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 $e \in \texttt{exprs}(V)$, 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 and variables $v_1', \ldots v_n'$ are pairwise different and not in V, 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 $\nu'(v_i') = \nu(v_i)$ for each $i : 1 \leq i \leq n$, 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 $X \to \mathbb{N}$. 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 $S = \{s_1, s_2, \ldots\}$ of shared variables and a set $L = \{l_1, l_2 \ldots\}$ of local variables. Each transition $(lc \blacktriangleright lc' : stmt)$ involves two locations lc and lc' and a statement $stmt$. We let Loc 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}. Program syntax is given in terms of pairwise different variables $v_1, \ldots v_n$ in $S \cup L$, expressions $e_1, \ldots e_n$ in $\mathtt{exprs}(S \cup L)$ and predicate π in $\mathtt{preds}(\mathtt{exprs}(S \cup L))$.

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

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* $\sigma : S \to \mathbb{Z}$ is a mapping that associates an integer value 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* $\eta : L \to \mathbb{Z}$ maps each local variable to an integer value. 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 $\overset{stmt}{\underset{P}{\rightarrowtail}}$ in order to define statements semantics (Fig. 3). We write $(\sigma, \eta, m) \overset{stmt}{\underset{P}{\rightarrowtail}} (\sigma', \eta', m')$, 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*).

$$\frac{(\sigma,\eta,m) \xrightarrow[P]{stmt} (\sigma',\eta',m')}{(\sigma,(lc,\eta)\oplus m) \xrightarrow[P]{(lc \blacktriangleright lc':stmt)} (\sigma',(lc',\eta')\oplus m')} : trans \qquad \frac{val_{\sigma,\eta}(\pi)}{(\sigma,\eta,m) \xrightarrow[P]{\pi} (\sigma,\eta,m)} : assume$$

$$\frac{(\sigma,\eta,m) \xrightarrow[P]{stmt} (\sigma',\eta',m') \quad (\sigma',\eta',m') \xrightarrow[P]{stmt'} (\sigma'',\eta'',m'')}{(\sigma,\eta,m) \xrightarrow[P]{stmt;stmt'} (\sigma'',\eta'',m'')} : seq \qquad \frac{m = ((lc_{ext},\eta')\oplus m')}{(\sigma,\eta,m) \xrightarrow[P]{join} (\sigma,\eta,m')} : join$$

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

Fig. 3. Semantics of concurrent programs.

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 $(\sigma_i,m_i) \xrightarrow[P]{t_{i+1}} (\sigma_{i+1},m_{i+1})$ for each $i\ :\ 0 \leq i < n$ and σ_0 and m_0 are initial. Each of the configurations (σ_i,m_i), for $i\ :\ 0 \leq i \leq n$, is then said to be *reachable*.

Counting Logic. We use $@Loc$ to mean the set $\{@lc \mid lc \in Loc\}$ of boolean variables. Intuitively, $@lc$ evaluates to $\texttt{tt}$ exactly when the process evaluating it is at location lc. We associate a *counting variable* $(\pi)^{\#}$ to each predicate π in $\texttt{preds}(@Loc,\texttt{exprs}(S\cup L))$. Intuitively, in a given program configuration, the variable $(\pi)^{\#}$ counts the number of processes for which the predicate π holds. We let $\Omega_{Loc,S,L}$ be the set $\left\{(\pi)^{\#} | \pi \in \texttt{preds}(@Loc,\texttt{exprs}(S\cup L))\right\}$. A *counting predicate* is any predicate in $\texttt{preds}(\texttt{exprs}(S\cup\Omega_{Loc,S,L}))$. Elements in $\texttt{exprs}(S\cup L)$ and $\texttt{preds}(@Loc,\texttt{exprs}(S\cup L))$ 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 the valuation $val_{\sigma,m}(v) = \sigma(v)$ for $v \in S$. Our counting logic is quite expressive. For instance, we can capture assertion violations, deadlocks or program invariants. For location lc, we let $enabled(lc)$ in $\texttt{preds}(\texttt{exprs}(S\cup L))$ define when a process can fire some transition from lc. The following counting predicates capture sets of configurations from Fig. 1.

$$\omega_{assert} = (proc@lc_4 \wedge \neg(prev \leq max))^{\#} \geq 1 \quad \omega_{inv} = (count = \textstyle\sum_{lc \in proc@Loc}(lc)^{\#})$$

$$\omega_{deadlock} = \textstyle\bigwedge_{lc \in proc@Loc \cup main@Loc}(lc \wedge enabled(lc))^{\#} = 0$$

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 [8, 9]) in order to generate an abstraction in the form of a boolean program $\texttt{abstOf}_{\Pi}(P) = \left(\tilde{S}, \tilde{L}, \tilde{T}\right)$ 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 bijection associates a predicate $\texttt{predOf}(\tilde{v})$ 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 $(lc \blacktriangleright lc' : \texttt{abstOf}_{\Pi}(stmt))$ with the same source and destination locations lc, lc', but with an abstracted statement $\texttt{abstOf}_{\Pi}(stmt)$ that may operate on the variables $\tilde{S} \cup \tilde{L}$. For instance, statement $(count := count + 1)$ in Fig. 1 is abstracted with the multiple assignment:

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

The value of the variable *count_leq_wait* after execution of the multiple assignment (1) is $\texttt{tt}$ if $\neg wait\_leq\_count \wedge count\_leq\_wait$ holds, $\texttt{ff}$ 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.

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

Syntax and semantics of boolean programs. We describe the syntax of boolean programs. Variables $\tilde{v}_1, \ldots, \tilde{v}_n$ are in $\tilde{S} \cup \tilde{L} \cup \tilde{L}_p$. Predicate π is in $\texttt{preds}(\tilde{S} \cup \tilde{L})$, and predicates $\pi_1, \ldots, \pi_n$ are in $\texttt{preds}(\tilde{S} \cup \tilde{L} \cup \tilde{L}_p)$. We further require for the multiple assignment that if $\tilde{v}_i \in \tilde{S} \cup \tilde{L}$ then $vars(\pi_i) \subseteq \tilde{S} \cup \tilde{L}$.

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

Apart from the variables being now boolean, the main difference between Fig. 4 and Fig. 3 is the *assign* statement. For this, we write $(\tilde{\sigma}, \tilde{\eta}, \tilde{\eta}_p) \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{\tilde{v}_1, \ldots \tilde{v}_n := \pi_1, \ldots \pi_n} (\tilde{\sigma}', \tilde{\eta}', \tilde{\eta}'_p)$ and 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 $\tilde{\eta}_{p,1} = \tilde{\eta}_p \left[\left\{\tilde{l} \leftarrow \tilde{l}_p | \tilde{l} \in \tilde{L}\right\}\right]$, 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. An $\texttt{abstOf}_{\Pi}(P)$ run is a sequence $(\tilde{\sigma}_0, \tilde{m}_0), \tilde{t}_1, ..., \tilde{t}_n, (\tilde{\sigma}_n, \tilde{m}_n)$. It is *feasible* if $(\tilde{\sigma}_i, \tilde{m}_i) \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{\tilde{t}_{i+1}} (\tilde{\sigma}_{i+1}, \tilde{m}_{i+1})$ for each $i \ : \ 0 \leq i < n$ and $\tilde{\sigma}_0$, $\tilde{m}_0$ are initial. Configurations $(\tilde{\sigma}_i, \tilde{m}_i)$, for $i : 0 \leq i \leq n$, are then said to be reachable.

$$\frac{(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{stmt} (\tilde{\sigma}', \tilde{\eta}', \tilde{m}')}{(\tilde{\sigma}, (lc, \tilde{\eta}) \oplus \tilde{m}) \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{(lc \blacktriangleright lc' : stmt)} (\tilde{\sigma}', (lc', \tilde{\eta}') \oplus \tilde{m}')} : trans \qquad \frac{val_{\tilde{\sigma}, \tilde{\eta}}(\pi)}{(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{\pi} (\tilde{\sigma}, \tilde{\eta}, \tilde{m})} : assume$$

$$\frac{(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{stmt} (\tilde{\sigma}', \tilde{\eta}', \tilde{m}') \text{ and } (\tilde{\sigma}', \tilde{\eta}', \tilde{m}') \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{stmt'} (\tilde{\sigma}'', \tilde{\eta}'', \tilde{m}'')}{(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{stmt;stmt'} (\tilde{\sigma}'', \tilde{\eta}'', \tilde{m}'')} : sequence$$

$$\frac{\tilde{m}' = (lc_{ent}, \tilde{\eta}_{init}) \oplus \tilde{m}}{(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{spawn} (\tilde{\sigma}, \tilde{\eta}, \tilde{m}')} : spawn \qquad \frac{\tilde{m} = \left((lc_{ext}, \tilde{\eta}') \oplus \tilde{m}'\right)}{(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{join} (\tilde{\sigma}, \tilde{\eta}, \tilde{m}')} : join$$

$$\frac{\begin{array}{c} \tilde{\sigma}' = \tilde{\sigma}[\left\{\tilde{v}_i \leftarrow val_{\tilde{\sigma}, \tilde{\eta}}(\pi_i) | \tilde{v}_i \in \tilde{S}\right\}] \\ \tilde{\eta}' = \tilde{\eta}[\left\{\tilde{v}_i \leftarrow val_{\tilde{\sigma}, \tilde{\eta}}(\pi_i) | \tilde{v}_i \in \tilde{L}\right\}] \\ h : \{1, ...|\tilde{m}|\} \to \{1, ...|\tilde{m}'|\} \text{ some bijection associating each } (lc_p, \tilde{\eta}_p)_i \in \tilde{m} \\ \text{to some } (lc_p, \tilde{\eta}'_p)_{h(i)} \in \tilde{m}' \text{ s.t. } (\tilde{\sigma}, \tilde{\eta}, \tilde{\eta}_p) \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{\tilde{v}_1, \ldots \tilde{v}_n, := \pi_1, \ldots \pi_n} (\tilde{\sigma}', \tilde{\eta}', \tilde{\eta}'_p) \end{array}}{(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{\tilde{v}_1, \ldots \tilde{v}_n := \pi_1, \ldots \pi_n} (\tilde{\sigma}', \tilde{\eta}', \tilde{m}')} : assign$$

Fig. 4. Semantics of boolean concurrent programs.

Relation between P and $\texttt{abstOf}_{\Pi}(P)$. Given a shared configuration $\tilde{\sigma}$, we let $\texttt{predOf}(\tilde{\sigma})$ denote the predicate $\bigwedge_{\tilde{s}\in\tilde{S}}(\tilde{\sigma}(\tilde{s}) \Leftrightarrow \texttt{predOf}(\tilde{s}))$. In a similar manner, we let $\texttt{predOf}(\tilde{\eta})$ denote $\bigwedge_{\tilde{l}\in\tilde{L}}(\tilde{\eta}(\tilde{l}) \Leftrightarrow \texttt{predOf}(\tilde{l}))$. Notice that $vars(\texttt{predOf}(\tilde{\sigma})) \subseteq S$ and $vars(\texttt{predOf}(\tilde{\eta})) \subseteq S \cup L$. We abuse notation and use $val_{\sigma}(\tilde{\sigma})$ (resp. $val_{\sigma,\eta}(\tilde{\eta})$) to mean that $val_{\sigma}(\texttt{predOf}(\tilde{\sigma}))$ (resp. $val_{\sigma,\eta}(\texttt{predOf}(\tilde{\eta}))$) holds. We also use $val_{\tilde{\sigma},\tilde{\eta}}(\pi)$, 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 $\texttt{predOf}(\tilde{v}) = \pi'$. We let $val_{\sigma,m}(\tilde{m})$ mean there is a bijection $h : \{1, ...|\tilde{m}|\} \to \{1, ...|\tilde{m}'|\}$ s.t. we can associate to each $(lc, \eta)_i$ in m an $(lc, \tilde{\eta})_{h(i)}$ in $\tilde{m}$ such that $val_{\sigma,\eta}(\tilde{\eta})$ for each $i : 1 \leq i \leq |m|$. The *concretization* of an $\texttt{abstOf}_{\Pi}(P)$ configuration $(\tilde{\sigma}, \tilde{m})$ is $\gamma((\tilde{\sigma}, \tilde{m})) = \{(\sigma, m) | val_{\sigma}(\tilde{\sigma}) \wedge val_{\sigma,m}(\tilde{m})\}$. The *abstraction* of (σ, m) is $\alpha((\sigma, m)) = \{(\tilde{\sigma}, \tilde{m}) | val_{\sigma}(\tilde{\sigma}) \wedge val_{\sigma,m}(\tilde{m})\}$. We initialize the $\texttt{abstOf}_{\Pi}(P)$ variables such that for each initial σ_{init}, m_{init} of P, there are $\tilde{\sigma}_{init}, \tilde{m}_{init}$ with $\alpha((\sigma_{init}, m_{init})) = \{(\tilde{\sigma}_{init}, \tilde{m}_{init})\}$. The abstraction $\alpha(\rho)$ of a P run $\rho = (\sigma_0, m_0), t_1, ...t_n, (\sigma_n, m_n)$ is the singleton set of P runs $\left\{(\tilde{\sigma}_0, \tilde{m}_0), \tilde{t}_1, ...\tilde{t}_n, (\tilde{\sigma}_n, \tilde{m}_n) | \alpha((\sigma_i, m_i)) = \{(\tilde{\sigma}_i, \tilde{m}_i)\} \text{ and } \tilde{t}_i = \texttt{abstOf}_{\Pi}(t_i)\right\}$.

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

5.2 Encoding into a counter machine

Assume a program $P = (S, L, T)$, a set $\Pi_0 \subseteq \texttt{preds}(\texttt{exprs}(S \cup L))$ of predicates and two counting predicates, an invariant ω_{inv} in $\texttt{preds}(\texttt{exprs}(S \cup \Omega_{Loc,S,L}))$ and a target ω_{trgt} in $\texttt{preds}(\texttt{exprs}(\Omega_{Loc,S,L}))$. We write $\texttt{abstOf}_{\Pi}(P) = \left(\tilde{S}, \tilde{L}, \tilde{T}\right)$ to mean the abstraction of P wrt. $\Pi = \cup_{(\pi)^{\#} \in vars(\omega_{inv}) \cup vars(\omega_{trgt})} atoms(\pi) \cup \Pi_0$. Intuitively, this step results in the formulation of a state reachability problem of a counter machine $enc(\texttt{abstOf}_{\Pi}(P))$ that captures reachability of abstractions of ω_{trgt} configurations with $\texttt{abstOf}_{\Pi}(P)$ runs that are strengthened wrt. ω_{inv}.

A 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 a set of initial 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 $\texttt{preds}(\texttt{exprs}(A \cup C))$ 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 $\texttt{exprs}(A \cup C)$ and pairwise different $c_1, \ldots c_n$ in C. We only write grd (resp. cmd) in case cmd is empty (resp. grd is $\texttt{tt}$) in $grd \Rightarrow cmd$.

A *machine configuration* is a pair (q, θ) where q is a state in Q and θ is a mapping $C \to \mathbb{N}$. 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 $(q_i, \theta_i) \xrightarrow[M]{\delta_{i+1}} (q_{i+1}, \theta_{i+1})$ for $i : 0 \leq i < n$. 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. $q_n = q_{trgt}$.

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

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

$$\frac{\exists A.val_\theta\,(\pi) \wedge \theta' = \theta\,[\{c_i \leftarrow val_\theta\,(e_i)\,|i : 1 \leq i \leq n\}]}{\theta \overset{grd \Rightarrow (c_1 \ldots c_n := e_1 \ldots e_n)}{\underset{M}{\rightarrowtail}} \theta'} : gcmd$$

Fig. 5. Semantics of a counter machine

Encoding. We describe in the following a counter machine $enc\,(\texttt{abstOf}_\Pi(P))$ obtained as an encoding of the boolean program $\texttt{abstOf}_\Pi(P)$. Recall $\texttt{abstOf}_\Pi(P)$ results from an abstraction (Def. 1) wrt. $\cup_{(\pi)^\# \in vars(\omega_{inv}) \cup vars(\omega_{trgt})} atoms(\pi) \cup \Pi_0$ of the concurrent program P. The machine $enc\,(\texttt{abstOf}_\Pi(P))$ is a tuple $(Q, C, \Delta, Q_{Init}, \Theta_{Init}, q_{trgt})$. Each state in Q is either the target state q_{trgt} or is associated to a shared configuration $\tilde{\sigma}$ of $\texttt{abstOf}_\Pi(P)$. We write $q_{\tilde{\sigma}}$ to make the association explicit. There is a bijection that associates a process configuration $(lc, \tilde{\eta})$ to each counter $c_{(lc,\tilde{\eta})}$ in C. Transitions Δ coincide with $\cup_{t \in \tilde{T}} \Delta_t \cup \Delta_{trgt}$ as described in Fig. 6. We abuse notation and associate to each statement *stmt* appearing in $\texttt{abstOf}_\Pi(P)$ the set $enc\,(stmt)$ of tuples $[(\tilde{\sigma}, \tilde{\eta}) : op : (\tilde{\sigma}', \tilde{\eta}')]_{stmt}$ generated in Fig. 6. Given a multiset $\tilde{m}$ of program configurations, we write $\theta_{\tilde{m}}$ to mean the mapping associating $\tilde{m}((lc, \tilde{\eta}))$ to each counter $c_{(lc,\tilde{\eta})}$ in C. We let Q_{Init} be the set $\{q_{\tilde{\sigma}} | \tilde{\sigma}$ is an initial shared state of $\texttt{abstOf}_\Pi(P)\}$, and Θ_{Init} be the set $\{\theta_{\tilde{m}} | \tilde{m}((lc_{ent}, \tilde{\eta})) = 1$ for an $\tilde{\eta}$ initial in $\texttt{abstOf}_\Pi(P)$ and 0 otherwise$\}$. We associate a program configuration $(\tilde{\sigma}, \tilde{m})$ to each machine configuration $(q_{\tilde{\sigma}}, \theta_{\tilde{m}})$. The machine encodes $\texttt{abstOf}_\Pi(P)$ in the following sense

Lemma 1. *q_{trgt} is $enc\,(\texttt{abstOf}_\Pi(P))$ reachable iff a configuration $(\tilde{\sigma}, \tilde{m})$ such that $\omega_{trgt}\left[\left\{(\pi)^\# \leftarrow \sum_{\{(lc,\tilde{\eta})|val_{\tilde{\sigma},(lc,\tilde{\eta})}(\pi)\}} \tilde{m}(lc, \tilde{\eta}) | (\pi)^\# \in vars(\omega_{trgt})\right\}\right]$ is reachable in $\texttt{abstOf}_\Pi(P)$.*

Observe that all transitions of a boolean program $\texttt{abstOf}_\Pi(P)$ are monotonic, i.e., if a configuration $(\tilde{\sigma}', \tilde{m}')$ is obtained from $(\tilde{\sigma}, \tilde{m})$ using a transition, then the same transition can obtain a configuration larger (i.e., has the same and possibly more processes) than $(\tilde{\sigma}', \tilde{m}')$ from any configuration larger than $(\tilde{\sigma}, \tilde{m})$. This reflects in the monotonicity of all transitions in Fig. 6 (except for rule *target*). Rule *target* results in monotonic machine transitions for all counting predicates ω_{trgt} that denote upward closed sets of processes. This is for instance the case of predicates capturing assertion violation but not of those capturing deadlocks (see Sec. 4). An encoding $enc\,(\texttt{abstOf}_{\Pi}(P))$ is said to be monotonic if all its transitions are monotonic. Checking program assertion violations always results in monotonic encodings.

Lemma 2. *State reachability of all monotonic encodings is decidable.*

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

$$\frac{}{(q_{\tilde{\sigma}} : \omega_{trgt}\left[\left\{(\pi)^{\#} \leftarrow \sum_{\left\{(lc,\tilde{\eta}) | val_{\tilde{\sigma},(lc,\tilde{\eta})}(\pi)\right\}} c_{((lc,\tilde{\eta}))} | (\pi)^{\#} \in vars(\omega_{trgt})\right\}\right] : q_{trgt}) \in \Delta_{trgt}} : target$$

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

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

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

$$\frac{\begin{array}{c}\tilde{\sigma}' = \tilde{\sigma}[\left\{\tilde{v}_i \leftarrow val_{\tilde{\sigma},\tilde{\eta}}(\pi_i) | \tilde{v}_i \in \tilde{S}\right\}] \qquad \tilde{\eta}' = \tilde{\eta}[\left\{\tilde{v}_i \leftarrow val_{\tilde{\sigma},\tilde{\eta}}(\pi_i) | \tilde{v}_i \in \tilde{L}\right\}] \\ B = \left\{a_{(lc,\tilde{\eta}_p),(lc,\tilde{\eta}'_p)} | lc \in Loc \text{ and } (\tilde{\sigma}, \tilde{\eta}, \tilde{\eta}_p) \xmapsto[\texttt{abstOf}_{\Pi}(P)]{\tilde{v}_1,\ldots\tilde{v}_n,:=\pi_1,\ldots\pi_n} (\tilde{\sigma}', \tilde{\eta}', \tilde{\eta}'_p)\right\}\end{array}}{\left[(\tilde{\sigma}, \tilde{\eta}) : \left(\begin{array}{c}\bigwedge_{(lc,\tilde{\eta}_p)}(c_{(lc,\tilde{\eta}_p)} = \sum_{a_{(lc,\tilde{\eta}_p),(lc,\tilde{\eta}'_p)} \in B} a_{(lc,\tilde{\eta}_p),(lc,\tilde{\eta}'_p)}) \\ \Rightarrow \cup_{(lc,\tilde{\eta}'_p)}\left\{c_{(lc,\tilde{\eta}'_p)} := \sum_{a_{(lc,\tilde{\eta}_p),(lc,\tilde{\eta}'_p)} \in B} a_{(lc,\tilde{\eta}_p),(lc,\tilde{\eta}'_p)}\right\}\end{array}\right) : (\tilde{\sigma}', \tilde{\eta}')\right]_{\tilde{v}_1,\ldots\tilde{v}_n,:=\pi_1,\ldots\pi_n}} : assign$$

Fig. 6. Encoding 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.

However, monotonic encodings correspond to coarse over-approximations. Intuitively, bad configurations (such as those where a deadlock occurs, or those obtained in a backward exploration for a barrier based program as described in the running example) are no more guaranteed to be upward closed. This loss of precision is irrevocable for techniques solely based on monotonic encodings. To regain some of the lost precision, we constrain the runs using counting invariants.

Lemma 3. *Any feasible P run has a feasible $\texttt{abstOf}_{\Pi}(P)$ run with a feasible run in any machine obtained as the strengthening of $enc\,(\texttt{abstOf}_{\Pi}(P))$ wrt. some P invariant $\omega_{inv} \in \texttt{preds}(\texttt{exprs}(S \cup \Omega_{Loc,S,L}))$.*

$$\frac{[q_{\tilde{\sigma}} : op : q_{\tilde{\sigma}'}] \in \Delta}{[q_{\tilde{\sigma}} : grd_{\tilde{\sigma}}(\omega_{inv}); op; grd_{\tilde{\sigma}'}(\omega_{inv}) : q_{\tilde{\sigma}'}] \in \Delta'} \; strengthen$$

Fig. 7. Strengthening of a transition of a counter machine $enc(\mathtt{abstOf}_{\Pi}(P))$ given a counting invariant ω_{inv} using the predicate $grd_{\tilde{\sigma}}(\omega_{inv}) = \exists S.\mathtt{predOf}(\tilde{\sigma}) \wedge \omega_{inv}\left[\left\{(\pi)^{\#} \leftarrow \sum_{\{(lc,\tilde{\eta})|val_{\tilde{\sigma},(lc,\tilde{\eta})}(\pi)\}} c_{((lc,\tilde{\eta}))}|(\pi)^{\#} \in vars(\omega_{inv})\right\}\right]$ in $\mathtt{preds}(\mathtt{exprs}(C))$.

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

Lemma 4. *State reachability is in general undecidable after strengthening.*

5.3 Constrained monotonic abstraction and preorder refinement

This step addresses the state reachability problem for a 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 encodings. The idea here [17] 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 run is identified and an interpolant I (with $vars(I) \subseteq C$) is used to refine the preorder as follows: $\preceq_{i+1} := \{(\theta, \theta')|\theta \preceq_i \theta' \wedge (val_{\theta}(I) \Leftrightarrow val_{\theta'}(I))\}$. Although stronger, the new preorder is again a well quasi ordering and the run is guaranteed to be eliminated in the next round. We refer the reader to [4] for more details.

Lemma 5 (CMA [4]). *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 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

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

Working := $\cup_{e \in \mathbf{Min}_{\preceq}(\mathbb{N}^{|C|})} \{((q_{trgt}, e), (q_{trgt}, e))\}$, Visited := $\{\}$;
while Working $\neq \{\}$ **do**
    $((q, \theta), \rho)$ =pick and remove a member from Working;
    Visited $\cup = \{((q, \theta), \rho)\}$;
    if $(q, \theta) \in Q_{Init} \times \Theta_{Init}$ **then return** ρ;
    foreach $\delta \in \Delta$ **do**
        $pre = \mathbf{Min}_{\preceq}(\mathbf{Pre}_{\delta}(\mathbf{Up}_{\preceq}((q, \theta))))$;
        foreach $(q', \theta') \in pre$ **do**
            if $\theta'' \preceq \theta'$ *for some* $((q', \theta''), \_)$ *in* Working $\cup$ Visited **then**
                continue;
            else
                foreach $((q', \theta''), \_) \in$ Working **do**
                    if $\theta' \preceq \theta''$ **then** Working = Working $\setminus \{((q', \theta'), \_)\}$;
                foreach $((q', \theta''), \_) \in$ Visited **do**
                    if $\theta' \preceq \theta''$ **then** Visited = Visited $\setminus \{((q', \theta'), \_)\}$;
                Working $\cup = \{((q', \theta'), (q', \theta'); \delta; \rho)\}$
return `not_reachable`;

Algorithm 1: Monotonic abstraction

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 run 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 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 (the problem is undecidable). 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 in this paper cannot express).

Table 1. Checking assertion violation with PACMAN

example	P	$enc\,(\mathtt{abstOf}_{\Pi}(P))$	outer loop num.	outer loop preds.	inner loop num.	inner loop preds.	results time(s)	results 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

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 understanding, the reported examples which require refinements of the natural preorder cannot be verified by techniques such as [6, 8]. Indeed, such approaches always adopt monotonic abstractions when the correctness of these examples crucially depends on the fact that non-monotonic behaviors of barriers are taken into account.

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 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 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 faulty runs as expected. The tuples under the P column respectively refer to the number of variables, procedures and transitions in the original program. The tuples under the $enc\,(\mathtt{abstOf}_{\Pi}(P))$ column refer to the number of counters, states and transitions in the extended counter machine.

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 helped our state space exploration algorithms. Still, our prototype did not terminate on several larger examples. We are working on improiving scalability by coming up and combining with more clever optimisations.

Table 2. Checking deadlock with PACMAN

example	P	$enc\,(\mathtt{abstOf}_{\Pi}(P))$	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 and 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 also aim to generalize to richer variable types.

Acknowledgments. The authors would like to thank the anonymous reviewers for their helpful remarks and relevant references.

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, A. Annichini, S. Bensalem, A. Bouajjani, P. Habermehl, and Y. Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 146–159. Springer, 1999.
3. P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. LICS '96,* 11^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 313–321, 1996.
4. 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.
5. K. Bansal, E. Koskinen, T. Wies, and D. Zufferey. Structural counter abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 62–77. Springer, 2013.
6. G. Basler, M. Hague, D. Kroening, C.-H. L. Ong, T. Wahl, and H. Zhao. Boom: Taking boolean program model checking one step further. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 145–149, Berlin, Heidelberg, 2010. Springer-Verlag.
7. L. E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *Amer. J. Math.*, 35:413–422, 1913.
8. A. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *Computer Aided Verification*, pages 356–371. Springer, 2011.
9. 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.
10. A. F. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 356–371. Springer, 2011.
11. A. Downey. *The Little Book of SEMAPHORES (2nd Edition): The Ins and Outs of Concurrency Control and Common Mistakes.* Createspace Independent Pub, 2009.
12. A. Farzan, Z. Kincaid, and A. Podelski. Proofs that count. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 151–164, New York, NY, USA, 2014. ACM.
13. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.
14. C. Flanagan and S. Qadeer. Thread-modular model checking. In T. Ball and S. K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 213–224. Springer, 2003.
15. 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.
16. A. Kaiser, D. Kroening, and T. Wahl. Lost in abstraction: Monotonicity in multithreaded programs. In P. Baldan and D. Gorla, editors, *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy,*

September 2-5, 2014. Proceedings, volume 8704 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2014.
17. 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 $lock := \mathtt{tt},\ writing := \mathtt{ff}$

<u>*main :*</u>
$lc_{ent} \blacktriangleright lc_{ent}$: $spawn(writer)$
$lc_{ent} \blacktriangleright lc_{ent}$: $readcount = 0 \wedge\ lock;\ spawn(reader);\ readcount := readcount + 1$
$lc_{ent} \blacktriangleright lc_{ent}$: $readcount! = 0; spawn(reader); readcount := readcount + 1$

<u>*reader :*</u>
$lc_{ent} \blacktriangleright lc_{err}$: $writing$
$lc_{ent} \blacktriangleright lc_{ext}$: $readcount = 1; readcount := readcount - 1; lock := \mathtt{tt}$
$lc_{ent} \blacktriangleright lc_{ext}$: $readcount! = 1; readcount := readcount - 1$

<u>*writer :*</u>
$lc_{ent} \blacktriangleright lc_1$: $lock; lock := \mathtt{ff}$
$lc_1 \blacktriangleright lc_2$: $writing := \mathtt{tt}$
$lc_2 \blacktriangleright lc_3$: $writing := \mathtt{ff}$
$lc_3 \blacktriangleright lc_{ext}$: $lock := \mathtt{tt}$

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 $allocated := \texttt{ff}$

<u>*main :*</u>

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

<u>*parent :*</u>

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

<u>*child :*</u>

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

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 $enough := \texttt{ff}, flag := *, barrierOpen := \texttt{ff}$

<u>main :</u>
$lc_{ent} \blacktriangleright lc_1 : \neg enough; spawn(proc); count := count + 1$
$lc_1 \blacktriangleright lc_{ent} : enough := \texttt{ff}$
$lc_1 \blacktriangleright lc_{ent} : enough := \texttt{tt}$

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

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

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

Fig. 11. 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 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

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.

int $count1 := 0, count2 := 0$
bool $enough := \texttt{ff}$

$\underline{main:}$
$lc_{ent} \blacktriangleright lc_1 \; : \; spawn(proc1); count1 := count1 + 1$
$lc_1 \blacktriangleright lc_{ent} \; : \; spawn(proc2); count2 := count2 + 1$
$lc_{ent} \blacktriangleright lc_2 \; : \; enough := \texttt{tt}$

$\underline{proc1:}$
$lc_{ent} \blacktriangleright lc_1 \; : \; enough$
$lc_1 \blacktriangleright lc_{err} \; : \; count1 \neq count2$

$\underline{proc2:}$
$lc_{ent} \blacktriangleright lc_1 \; : \; enough$

Fig. 12. As Many

int $wait := 0, count := 0, open := 0$
bool $proceed := \texttt{ff}$

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

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

Fig. 13. Buggy Barrier No.1

A.6 Barriers causing deadlock

In Fig.13 a buggy implementation of barrier is demonstrated. This example is based on an example in [11]. 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.

int $wait := 0, count := 0$
bool $proceed := \texttt{ff}$

<u>*main* :</u>
$lc_{ent} \blacktriangleright lc_{ent}$: $spawn(proc1); count := count + 1$
$lc_{ent} \blacktriangleright lc_{ent}$: $spawn(proc2)$
$lc_{ent} \blacktriangleright lc_{ext}$: $proceed := \texttt{tt}$

<u>*proc*1 :</u>
$lc_{ent} \blacktriangleright lc_1$: $wait := wait + 1$
$lc_1 \blacktriangleright lc_2$: $proceed \wedge wait = count$
$lc_1 \blacktriangleright lc_{err}$: $proceed \wedge wait \neq count \wedge (proc1@lc_{ent})^{\#} = 0$
<u>*proc*2 :</u>
$lc_{ent} \blacktriangleright lc_1$: $wait > 0; wait := wait - 1$

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*.

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

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

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

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

Fig. 15. Buggy Barrier No.3

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`

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

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

Fig. 16. Buggy Barrier in Loop

The example in Fig.16 is based on an example in [11]. 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.

B Proofs

In this section, assume a program $P = (S, L, T)$, a set $\Pi_0 \subseteq \texttt{preds}(\texttt{exprs}(S \cup L))$ of predicates and two counting predicates, namely an invariant predicate ω_{inv} in $\texttt{preds}(\texttt{exprs}(S \cup \Omega_{Loc,S,L}))$ and a target predicate ω_{trgt} belonging to $\texttt{preds}(\texttt{exprs}(\Omega_{Loc,S,L}))$. We write $\texttt{abstOf}_{\Pi}(P) = \left(\tilde{S}, \tilde{L}, \tilde{T}\right)$ to mean the abstraction of P wrt. $\Pi = \cup_{(\pi)^{\#} \in vars(\omega_{inv}) \cup vars(\omega_{trgt})} atoms(\pi) \cup \Pi_0$. We write $enc\left(\texttt{abstOf}_{\Pi}(P)\right) = (Q, C, \Delta, Q_{Init}, \Theta_{Init}, q_{trgt})$ to mean the counter machine encoding $\texttt{abstOf}_{\Pi}(P)$.

In order to prove Lem. 1, we first establish Lem. 6. Intuitively, the lemma relates the semantics of the statements of a boolean program to the one of the operations of its encoding. Recall $enc\,(stmt)$ is the set of tuples $[(\tilde{\sigma}, \tilde{\eta}) : op : (\tilde{\sigma}, \tilde{\eta})]_{stmt}$ generated in Fig. 6 during the encoding of the statement *stmt* of $\texttt{abstOf}_{\Pi}(P)$.

Lemma 6. *For any statement stmt appearing in* $\texttt{abstOf}_{\Pi}(P)$, $(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{stmt} (\tilde{\sigma}', \tilde{\eta}', \tilde{m}')$ *iff* $\theta_{\tilde{m}} \xrightarrow[enc(\texttt{abstOf}_{\Pi}(P))]{op} \theta_{\tilde{m}'}$ *for some* $[(\tilde{\sigma}, \tilde{\eta}) : op : (\tilde{\sigma}', \tilde{\eta}')]_{stmt}$ *in* $enc\,(stmt)$.

Proof. We proceed by induction on the number of atomic statements (i.e., assume, spawn, join or assign statements) appearing in *stmt*.

Base case, *stmt* consists of the atomic statement:

1. π is an assume statement appearing in $\texttt{abstOf}_{\Pi}(P)$. The semantics of boolean programs in Fig.4 ensures that $(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{\pi} (\tilde{\sigma}', \tilde{\eta}', \tilde{m}')$ iff $val_{\tilde{\sigma},\tilde{\eta}}\,(\pi)$ holds, $\tilde{\sigma} = \tilde{\sigma}'$, $\tilde{\eta} = \tilde{\eta}'$ and $\tilde{m} = \tilde{m}'$. In addition, the definition of the encoding of a boolean program in Fig. 6 only generates $[\_ : op : \_]_{\pi}$ for $op = nop$. It ensures that $[(\tilde{\sigma}, \tilde{\eta}) : nop : (\tilde{\sigma}', \tilde{\eta}')]_{stmt}$ is generated iff $val_{\tilde{\sigma},\tilde{\eta}}\,(\pi)$, $\tilde{\sigma} = \tilde{\sigma}'$ and $\tilde{\eta} = \tilde{\eta}'$. Finally, counter machines semantics in Fig. 5 ensures that $\theta_{\tilde{m}} \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{nop} \theta_{\tilde{m}}$ for any multiset $\tilde{m}$.
2. *spawn* is a statement appearing in $\texttt{abstOf}_{\Pi}(P)$. The semantics of boolean programs in Fig.4 ensure that $(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{spawn} (\tilde{\sigma}', \tilde{\eta}', \tilde{m}')$ iff $\tilde{\sigma} = \tilde{\sigma}'$, $\tilde{\eta} = \tilde{\eta}'$ and $\tilde{m}' = (lc_{ent}, \tilde{\eta}_{init}) \oplus \tilde{m}$ for each initial $\tilde{\eta}_{init}$. In addition, the definition of the encoding of a boolean program in Fig. 6 ensures that $[(\tilde{\sigma}, \tilde{\eta}) : op : (\tilde{\sigma}', \tilde{\eta}')]_{spawn}$ is generated iff $\tilde{\sigma} = \tilde{\sigma}'$, $\tilde{\eta} = \tilde{\eta}'$ and $op = (c_{(lc_{ent},\tilde{\eta}_{init})} := c_{(lc_{ent},\tilde{\eta}_{init})} + 1)$ for each initial $\tilde{\eta}_{init}$. Finally, counter machines semantics in Fig. 5 ensure that $\theta_{\tilde{m}} \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{c_{(lc_{ent},\tilde{\eta}_{init})} := (c_{(lc_{ent},\tilde{\eta}_{init})}+1)} \theta_{(lc_{ent},\tilde{\eta}_{init}) \oplus \tilde{m}}$ for any multiset $\tilde{m}$.

3. *join* is a statement appearing in $\texttt{abstOf}_\Pi(P)$. The semantics of boolean programs in Fig.4 ensure that $(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \xrightarrow[\texttt{abstOf}_\Pi(P)]{join} (\tilde{\sigma}', \tilde{\eta}', \tilde{m}')$ iff $\tilde{\sigma} = \tilde{\sigma}'$, $\tilde{\eta} = \tilde{\eta}'$ and $\tilde{m} = (lc_{ext}, \tilde{\eta}_1) \oplus \tilde{m}'$. In addition, the definition of the encoding of a boolean program in Fig. 6 ensures that $[(\tilde{\sigma}, \tilde{\eta}) : op : (\tilde{\sigma}', \tilde{\eta}')]_{join}$ is generated iff $\tilde{\sigma} = \tilde{\sigma}'$, $\tilde{\eta} = \tilde{\eta}'$ and $op = c_{(lc_{ext}, \tilde{\eta}_1)} \geq 1 \Rightarrow (c_{(lc_{ext}, \tilde{\eta}_1)} := c_{(lc_{ext}, \tilde{\eta}_1)} - 1)$. Finally, Fig. 5 ensures that $\theta_{(lc,\tilde{\eta}) \oplus \tilde{m}} \xrightarrow[\texttt{abstOf}_\Pi(P)]{c_{(lc,\tilde{\eta})} \geq 1 \Rightarrow (c_{(lc,\tilde{\eta})} := c_{(lc,\tilde{\eta})} - 1)} \theta_{\tilde{m}}$ for any multiset $\tilde{m}$ and program configuration $(lc, \tilde{\eta})$.
4. an *assign* statement $\tilde{v}_1, \dots \tilde{v}_n := \pi_1, \dots \pi_n$ appears in $\texttt{abstOf}_\Pi(P)$. The semantics of boolean programs in Fig.4 ensures that $(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \xrightarrow[\texttt{abstOf}_\Pi(P)]{\tilde{v}_1, \dots \tilde{v}_n := \pi_1, \dots \pi_n} (\tilde{\sigma}', \tilde{\eta}', \tilde{m}')$ iff $\tilde{\sigma}' = \tilde{\sigma}[\left\{\tilde{v}_i \leftarrow val_{\tilde{\sigma},\tilde{\eta}}(\pi_i) \,|\tilde{v}_i \in \tilde{S}\right\}]$, $\tilde{\eta}' = \tilde{\eta}[\left\{\tilde{v}_i \leftarrow val_{\tilde{\sigma},\tilde{\eta}}(\pi_i) \,|\tilde{v}_i \in \tilde{L}\right\}]$ and there is a bijection $h : \{1, ...|\tilde{m}|\} \to \{1, ...|\tilde{m}'|\}$ such that each $(lc_p, \tilde{\eta}_p)_i \in \tilde{m}$ is associated to a $(lc_p, \tilde{\eta}'_p)_{h(i)} \in \tilde{m}'$ with $(\tilde{\sigma}, \tilde{\eta}, \tilde{\eta}_p) \xrightarrow[\texttt{abstOf}_\Pi(P)]{\tilde{v}_1, \dots \tilde{v}_n, := \pi_1, \dots \pi_n} (\tilde{\sigma}', \tilde{\eta}', \tilde{\eta}'_p)$. In addition, the definition of the encoding of an assignment in Fig. 6 ensures that $[(\tilde{\sigma}, \tilde{\eta}) : op : (\tilde{\sigma}', \tilde{\eta}')]_{\tilde{v}_1, \dots \tilde{v}_n, := \pi_1, \dots \pi_n}$ are exactly generated when $\tilde{\sigma}' = \tilde{\sigma}[\left\{\tilde{v}_i \leftarrow val_{\tilde{\sigma},\tilde{\eta}}(\pi_i) \,|\tilde{v}_i \in \tilde{S}\right\}]$ and $\tilde{\eta}' = \tilde{\eta}[\left\{\tilde{v}_i \leftarrow val_{\tilde{\sigma},\tilde{\eta}}(\pi_i) \,|\tilde{v}_i \in \tilde{L}\right\}]$ and $op = (grd \Rightarrow cmd)$ is defined wrt. the set of auxiliary natural variables $A = \left\{a_{(lc,\tilde{\eta}_p),(lc,\tilde{\eta}'_p)} | lc \in Loc \text{ and } (\tilde{\sigma}, \tilde{\eta}, \tilde{\eta}_p) \xrightarrow[\texttt{abstOf}_\Pi(P)]{\tilde{v}_1, \dots \tilde{v}_n, := \pi_1, \dots \pi_n} (\tilde{\sigma}', \tilde{\eta}', \tilde{\eta}'_p)\right\}$ in the following way:

$$grd = \bigwedge_{(lc,\tilde{\eta}_p)} (c_{(lc,\tilde{\eta}_p)} = \sum_{a_{(lc,\tilde{\eta}_p),(lc,\tilde{\eta}'_p)} \in A} a_{(lc,\tilde{\eta}_p),(lc,\tilde{\eta}'_p)})$$
$$cmd = \cup_{(lc,\tilde{\eta}'_p)} \left\{c_{(lc,\tilde{\eta}'_p)} := \sum_{a_{(lc,\tilde{\eta}_p),(lc,\tilde{\eta}'_p)} \in A} a_{(lc,\tilde{\eta}_p),(lc,\tilde{\eta}'_p)}\right\}$$

Now, given two multisets $\tilde{m}, \tilde{m}'$, Fig. 5 ensures that $\theta_{\tilde{m}} \xrightarrow[enc(\texttt{abstOf}_\Pi(P))]{grd \Rightarrow cmd} \theta_{\tilde{m}'}$ iff there is a mapping $\gamma : A \to \mathbb{N}$ such that $val_{\theta_{\tilde{m}},\gamma}(grd)$ evaluates to true and $\theta_{\tilde{m}'} = \theta_{\tilde{m}} \left[c_{(lc,\tilde{\eta}'_p)} \leftarrow \sum_{a_{(lc,\tilde{\eta}_p),(lc,\tilde{\eta}'_p)} \in A} \gamma(a_{(lc,\tilde{\eta}_p),(lc,\tilde{\eta}'_p)})\right]$. Each element $a_{(lc,\tilde{\eta}_p),(lc,\tilde{\eta}'_p)}$ in A can be regarded as the number of $(lc, \tilde{\eta}_p)$ elements in $\tilde{m}$ that are sent to $(lc, \tilde{\eta}'_p)$ elements in $\tilde{m}'$. The guard ensures that each element in $\tilde{m}$ is sent to some elements in $\tilde{m}'$, and the command ensures that each element in $\tilde{m}$ comes from some element in $\tilde{m}$. This is possbile iff there is a bijection $h : \{1, ...|\tilde{m}|\} \to \{1, ...|\tilde{m}'|\}$ such that each $(lc_p, \tilde{\eta}_p)_i \in \tilde{m}$ is associated to a $(lc_p, \tilde{\eta}'_p)_{h(i)} \in \tilde{m}'$ with $(\tilde{\sigma}, \tilde{\eta}, \tilde{\eta}_p) \xrightarrow[\texttt{abstOf}_\Pi(P)]{\tilde{v}_1, \dots \tilde{v}_n, := \pi_1, \dots \pi_n} (\tilde{\sigma}', \tilde{\eta}', \tilde{\eta}'_p)$.

Suppose that for any statement *stmt* appearing in $\texttt{abstOf}_\Pi(P)$ of length smaller or equal to $n \geq 1$: $(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \xrightarrow[\texttt{abstOf}_\Pi(P)]{stmt} (\tilde{\sigma}', \tilde{\eta}', \tilde{m}')$ iff $\theta_{\tilde{m}} \xrightarrow[enc(\texttt{abstOf}_\Pi(P))]{op} \theta_{\tilde{m}'}$ for some $[(\tilde{\sigma}, \tilde{\eta}) : op : (\tilde{\sigma}', \tilde{\eta}')]_{stmt}$ in $enc(stmt)$. Using the base case, we know that for any $stmt'$ of length 1, $(\tilde{\sigma}', \tilde{\eta}', \tilde{m}') \xrightarrow[\mathtt{abstOf}_{\Pi}(P)]{stmt'} (\tilde{\sigma}'', \tilde{\eta}'', \tilde{m}'')$ iff $\theta_{\tilde{m}'} \xrightarrow[enc(\mathtt{abstOf}_{\Pi}(P))]{op'} \theta_{\tilde{m}''}$ for some $[(\tilde{\sigma}', \tilde{\eta}') : op' : (\tilde{\sigma}'', \tilde{\eta}'')]_{stmt'}$ in $enc\,(stmt')$.

The only way to build compound rules for boolean programs and their encodings are respectively the two sequence rules in Fig. 4 and in Fig. 6. We get that for any statement $stmt$ appearing in $\mathtt{abstOf}_{\Pi}(P)$ of length smaller or equal to $n+1$: $(\tilde{\sigma}, \tilde{\eta}, \tilde{m}) \xrightarrow[\mathtt{abstOf}_{\Pi}(P)]{stmt;stmt'} (\tilde{\sigma}'', \tilde{\eta}'', \tilde{m}'')$ iff $\theta_{\tilde{m}} \xrightarrow[enc(\mathtt{abstOf}_{\Pi}(P))]{op;op'} \theta_{\tilde{m}''}$ for some $[(\tilde{\sigma}, \tilde{\eta}) : op : (\tilde{\sigma}', \tilde{\eta}')]_{stmt}$ and $[(\tilde{\sigma}', \tilde{\eta}') : op' : (\tilde{\sigma}'', \tilde{\eta}'')]_{stmt'}$ in $enc\,(stmt)$ and $enc\,(stmt')$ respectively. □

Now we can prove Lem. 1:

Lemma 1. *q_{trgt} is $enc\,(\mathtt{abstOf}_{\Pi}(P))$ reachable iff a configuration $(\tilde{\sigma}, \tilde{m})$ such that $\omega_{trgt}\left[\left\{(\pi)^{\#} \leftarrow \sum_{\{(lc,\tilde{\eta})|val_{(lc,\tilde{\eta})}(\pi)\}} \tilde{m}((lc,\tilde{\eta}))|(\pi)^{\#} \in vars(\omega_{trgt})\right\}\right]$ is reachable in $\mathtt{abstOf}_{\Pi}(P)$.*

Proof. We prove that an $\mathtt{abstOf}_{\Pi}(P)$ run $\rho_{\mathtt{abstOf}_{\Pi}(P)} = (\tilde{\sigma}_0, \tilde{m}_0), \tilde{t}_1, \ldots(\tilde{\sigma}_n, \tilde{m}_n)$ is feasible with $\omega_{trgt}\left[\left\{(\pi)^{\#} \leftarrow \sum_{\{(lc,\tilde{\eta})|val_{(lc,\tilde{\eta})}(\pi)\}} \tilde{m}_n((lc,\tilde{\eta}))|(\pi)^{\#} \in vars(\omega_{trgt})\right\}\right]$, iff some $\rho_{enc(\mathtt{abstOf}_{\Pi}(P))} = (q_{\tilde{\sigma}_0}, \theta_{\tilde{m}_0}), \delta_1, \ldots (q_{\tilde{\sigma}_n}, \theta_{\tilde{m}_n}), \delta_{trgt}, (q_{trgt}, e)$ satisfying $\delta_{trgt} \in \Delta_{trgt}$ and $\delta_i \in \Delta_{\tilde{t}_i}$ as defined in Fig. 6 for each $\tilde{t}_i = (lc \blacktriangleright lc' : stmt)$ with $i : 1 \le i \le n$ is $enc\,(\mathtt{abstOf}_{\Pi}(P))$ feasible.

By definition $\tilde{\sigma}_0$ and $\tilde{m}_0$ are initial iff $q_{\tilde{\sigma}_0}$ and $\theta_{\tilde{m}_0}$ are also initial. In addition, the semantics of boolean programs in Fig. 4 ensure that $(\tilde{\sigma}, \tilde{m}) \xrightarrow[\mathtt{abstOf}_{\Pi}(P)]{(lc \blacktriangleright lc':stmt)} (\tilde{\sigma}', \tilde{m}')$ iff $\tilde{m} = (lc, \tilde{\eta}) \oplus \tilde{m}_1$ and $\tilde{m}' = (lc', \tilde{\eta}') \oplus \tilde{m}_1'$ and $(\tilde{\sigma}, \tilde{\eta}, \tilde{m}_1) \xrightarrow[\mathtt{abstOf}_{\Pi}(P)]{stmt} (\tilde{\sigma}', \tilde{\eta}', \tilde{m}_1')$. Using Lem. 6, we can show that: $(\tilde{\sigma}, \tilde{\eta}, \tilde{m}_1) \xrightarrow[\mathtt{abstOf}_{\Pi}(P)]{stmt} (\tilde{\sigma}', \tilde{\eta}', \tilde{m}_1')$ iff $\theta_{\tilde{m}_1} \xrightarrow[enc(\mathtt{abstOf}_{\Pi}(P))]{op} \theta_{\tilde{m}_1'}$ for some $[(\tilde{\sigma}, \tilde{\eta}) : op : (\tilde{\sigma}', \tilde{\eta}')]_{stmt}$ in $enc\,(stmt)$. Observe that $\theta_{\tilde{m}_1} \xrightarrow[enc(\mathtt{abstOf}_{\Pi}(P))]{op} \theta_{\tilde{m}_1'}$ is equivalent to $\theta_{(lc,\tilde{\eta}) \oplus \tilde{m}_1} \xrightarrow[enc(\mathtt{abstOf}_{\Pi}(P))]{c_{(lc,\tilde{\eta})} \ge 1 \Rightarrow (c_{(lc,\tilde{\eta})})^{--}; op; (c_{(lc',\tilde{\eta}')})^{++}} \theta_{(lc',\tilde{\eta}') \oplus \tilde{m}_1'}$. The rule $\delta = (q_{\tilde{\sigma}} : c_{(lc,\tilde{\eta})} \ge 1 \Rightarrow (c_{(lc,\tilde{\eta})})^{--}; op; (c_{(lc',\tilde{\eta}')})^{++} : q_{\tilde{\sigma}'})$ is generated in $\Delta_{(lc \blacktriangleright lc':stmt)}$ by the transition rule in Fig. 6. The semantics of counter machines in Fig. 5, gives that: $(q_{\tilde{\sigma}}, \theta_{\tilde{m}}) \xrightarrow[enc(\mathtt{abstOf}_{\Pi}(P))]{\delta} (q_{\tilde{\sigma}'}, \theta_{\tilde{m}'})$. For the last step, observe that: $\omega_{trgt}\left[\left\{(\pi)^{\#} \leftarrow \sum_{\{(lc,\tilde{\eta})|val_{(lc,\tilde{\eta})}(\pi)\}} \tilde{m}_n((lc,\tilde{\eta}))|(\pi)^{\#} \in vars(\omega_{trgt})\right\}\right]$ holds iff $val_{\theta_{\tilde{m}_n}}\left(\omega_{trgt}\left[\left\{(\pi)^{\#} \leftarrow \sum_{\{(lc,\tilde{\eta})|val_{(lc,\tilde{\eta})}(\pi)\}} c_{((lc,\tilde{\eta}))}|(\pi)^{\#} \in vars(\omega_{trgt})\right\}\right]\right)$ holds. □

Lemma 2. *State reachability of all encodings obtained with monotonic target counting predicates is decidable.*

Proof. First, notice that $(\Theta, \preceq)$, where Θ is the set of counter valuations and where $\theta \preceq \theta'$ (for $\theta, \theta' \in \Theta$) iff $\theta(c) \leq \theta'(c)$ for each $c \in C$ is a well quasi ordering wrt. $\sqsubseteq$ (defined by $(q, \theta) \sqsubseteq (q', \theta')$ iff $q = q'$ and $\theta \preceq \theta'$) [7]. We show in the following that encodings obtained with monotonic target counting predicates result in well structured transition systems [3, 13]. It is well known that state reachability is decidable for such systems. For this, we need to show that:

1. Except for the transitions generated by the target rule, each transition $(q : op : q')$ generated in Fig. 6 is monotonic wrt. the well quasi ordering $\sqsubseteq$. Established by Lem. 7.
2. The upward closure $\mathbf{Up}_{\sqsubseteq}((q, \theta)) = \{(q', \theta') | (q, \theta) \sqsubseteq (q', \theta')\}$ of a machine configuration (q, θ) can be represented as a Presburger definable set of machine configurations. In addition, for each Presburger definable set of machine configurations S (we write $\texttt{predOf}(S)$ to mean the associated Presburger formula), we can compute a finite set of minimal elements $\mathbf{Min}_{\sqsubseteq}(S) = \{(q_1, \theta_1), \ldots (q_n, \theta_n)\}$ (i.e., $(q_i, \theta_i) \not\sqsubseteq (q_j, \theta_j)$ if $i \neq j$ and for each $(q, \theta) \in S$ there is a $(q_i, \theta_i) \in \mathbf{Min}_{\preceq}(S)$ with $(q_i, \theta_i) \sqsubseteq (q, \theta)$). Established by Lem. 8.
3. The predecessors $\mathbf{Pre}_{(q:op:q')}(\mathbf{Up}_{\sqsubseteq}((q', \theta')))$ are effectively representable as a Presburger formula for each (q', θ') and transition $(q : op : q')$ generated in Fig. 6. Established by Lem. 9.

□

Lemma 7. *Transitions $(q : op : q')$ generated by all rules in Fig. 6, except for the target rule, are monotonic wrt. $\sqsubseteq$.*

Proof. Let *op* be some operation appearing in a generated transition $(q : op : q')$ of $enc(\texttt{abstOf}_{\Pi}(P))$. We say that an operation *op* is monotonic wrt. $\preceq$ if for each $\theta_1, \theta_2, \theta_3$ s.t. $\theta_1 \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{op} \theta_2$ and $\theta_1 \preceq \theta_3$ there exists an θ_4 s.t. $\theta_3 \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{op} \theta_4$ and $\theta_2 \preceq \theta_4$. Observe that $(q : op : q')$ is monotonic wrt. $\sqsubseteq$ iff *op* is monotonic wrt. $\preceq$. In addition, observe that if both *op* and *op'* are monotonic, then so is *op*; *op'*. It is therefore enough to show monotonicity of *nop*, $c \geq 1$, $(c)^{++}$, $(c)^{--}$ and $\bigwedge_{c_i \in C}(c_i = \sum_{a_{i,j} \in A} a_{i,j}) \Rightarrow \cup_j \left\{c_j := \sum_{a_{i,j} \in A} a_{i,j}\right\}$ where $A = \{a_{i,j} |$ for each $c_i \in C$ there is at least one $c_j \in C$ s.t. $a_{i,j} \in A\}$ is a set of auxiliary natural variables. The first four cases are straightforward. We show $grd \Rightarrow cmd$ is monotonic. Suppose we are given $\theta_1, \theta_2, \theta_3$ s.t. $\theta_1 \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{grd \Rightarrow cmd} \theta_2$ and $\theta_1 \preceq \theta_3$. We exhibit a θ_4 s.t. $\theta_3 \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{grd \Rightarrow cmd} \theta_4$ and $\theta_2 \preceq \theta_4$.

Since $\theta_1 \xrightarrow[\texttt{abstOf}_{\Pi}(P)]{grd \Rightarrow cmd} \theta_2$, we get that there is a valuation $\gamma : A \to \mathbb{N}$ s.t. $\theta_1(c_{(i)}) = \sum_{js.t.a_{i,j} \in A} \gamma(a_{i,j})$ and $\theta_2(c_j) = \sum_{is.t.a_{i,j} \in A} \gamma(a_{i,j})$.

If $\theta_1 \preceq \theta_3$ we have that for all $c_{(i)} \in C$, $\theta_1(c_{(i)}) \leq \theta_3(c_{(i)})$, thus $\theta_3(c_{(i)}) = \theta_1(c_{(i)}) + e_i = \sum_{js.t.a_{i,j} \in A} a_{i,j} + e_i$ where $e_i \geq 0$. By definition of A, for each c_i there is at least a j_0 such that $a_{i,j_0} \in A$.

We define $\theta_4(c_{(j_0)}) := \sum_{l s.t. a_{l,j_0} \in A} \gamma'(a_{l,j_0})$ where $\gamma' a_{l,j_0} = \gamma(a_{l,j_0}) + \delta_{li} e_i$ where δ_{li} is the Kronecker delta. So, $\theta_2(c_{(j_0)}) \preceq \theta_4(c_{(j_0)})$. We repeat the the process for each counter in C. This results in a θ_4 where $\theta_2 \preceq \theta_4$ and for which the transition is possible using the mapping γ'. □

Lemma 8. *The upward closure $\boldsymbol{Up}_{\sqsubseteq}((q,\theta)) = \{(q',\theta')|(q,\theta) \sqsubseteq (q',\theta')\}$ of a machine configuration (q,θ) can be represented as a Presburger definable set of machine configurations. In addition, for each Presburger definable set of machine configurations S (we write $\texttt{predOf}(S)$ to mean the associated Presburger formula), we can compute a finite set of minimal elements $\boldsymbol{Min}_{\sqsubseteq}(S) = \{(q_1,\theta_1), \ldots (q_n,\theta_n)\}$ (i.e., $(q_i,\theta_i) \not\sqsubseteq (q_j,\theta_j)$ if $i \neq j$ and for each $(q,\theta) \in S$ there is a $(q_i,\theta_i) \in \boldsymbol{Min}_{\preceq}(S)$ with $(q_i,\theta_i) \sqsubseteq (q,\theta)$).*

Proof. We use variable *state* to refer to the state of a machine configuration. We represent a machine configuration (q,θ) with the Presburger formula $\phi_{(q,\theta)}(state, C) = (state = q) \wedge \wedge_{c \in C} \theta(c) = c$. We define the Presburger formula $\phi_{\mathbf{Up}_{\sqsubseteq}((q,\theta))}(state, C) = (state = q) \wedge \wedge_{c \in C} \theta(c) \leq c$ to represent all machine configurations (q',θ') that satisfy it, i.e., $q' = q$ and $\theta(c) \preceq \theta'(c)$ for each $c \in C$, i.e., the upward closure of (q,θ). Given a set S that contains exactly all machine configurations that satisfy the formula $\texttt{predOf}(S)$ over $c_1, \ldots c_{|C|}$, we define the formula $\phi_{\mathbf{Min}_{\sqsubseteq}(S)} = (\texttt{predOf}(S) \wedge \forall c'_1, \ldots, c'_{|C|}.((\texttt{predOf}(S)\,[\{c \leftarrow c' | c \in C\}])) \Rightarrow ((\wedge_{c \in C} c = c' \wedge state = state') \vee \vee_{c \in C} c' \not\leq c \vee state \neq state'))$. Intuitively, a machine configuration (q,θ) satisfies $\phi_{\mathbf{Min}_{\sqsubseteq}(S)}$ iff it is in S and for every machine configuration (q',θ') also in S, either $(q,\theta) = (q',\theta')$ or $(q',\theta') \not\sqsubseteq (q,\theta)$. In other words, there is no other machine configuration in S that is both smaller and different. By well quasi ordering of $\sqsubseteq$, the number of machine configurations that satisfy $\phi_{\mathbf{Min}_{\sqsubseteq}(S)}$ is finite.

Lemma 9. *The set of predecessors $\boldsymbol{Pre}_{(q:op:q')}(S)$ is effectively representable as a Presburger formula for each Presburer representable set S of machine configurations and transition $(q : op : q')$ generated in Fig. 6.*

Proof. Given a transition $(q : op : q')$ the set $\mathbf{Pre}_{(q:op:q')}(S)$ can be represented with $\exists state, c'_1, \ldots, c'_{|C|}\,((state = q) \wedge \phi_{op} \wedge (state' = q') \wedge S\,[\{state \leftarrow state'\} \cup \{c \leftarrow c' | c \in C\}])$ where ϕ_{op} is a formula over C, C' defined for each operation as follows:

1. $op = nop$, then $\phi_{op} = \wedge_{c \in C} c = c'$.
2. $op = (grd \Rightarrow c_1, \ldots, c_n := e_1, \ldots, e_n)$, then $\phi_{op} = \exists A.(grd \wedge c_1 = e_1 \ldots \wedge c_n = e_n \wedge \wedge_{c \in C \setminus c_1, \ldots c_n} c = c')$.
3. $op = op_1; op_2$ then $\phi_{op} = \exists c''_1, \ldots c''_{|C|}.\phi_{op_1}\,[\{c' \leftarrow c'' | c \in C\}] \wedge \phi_{op_2}\,[\{c \leftarrow c'' | c \in C\}]$

□

In order to establish Lem. 3, we show two intermediate results Lem. 10 and Lem. 11.

First recall we assume a program $P = (S, L, T)$, a set $\Pi_0 \subseteq \texttt{preds}(\texttt{exprs}(S \cup L))$ of predicates and two counting predicates, namely an invariant predicate ω_{inv} in $\texttt{preds}(\texttt{exprs}(S \cup \Omega_{Loc,S,L}))$ and a target predicate ω_{trgt} belonging to $\texttt{preds}(\texttt{exprs}(\Omega_{Loc,S,L}))$.

We write $\texttt{abstOf}_{\Pi}(P) = \left(\tilde{S}, \tilde{L}, \tilde{T}\right)$ to mean the abstraction of P wrt. $\Pi = \cup_{(\pi)^{\#} \in vars(\omega_{inv}) \cup vars(\omega_{trgt})} atoms(\pi) \cup \Pi_0$.

Given a shared configuration $\tilde{\sigma}$, we write $\texttt{predOf}(\tilde{\sigma})$ to mean the predicate $\bigwedge_{\tilde{s} \in \tilde{S}} (\tilde{\sigma}(\tilde{s}) \Leftrightarrow \texttt{predOf}(\tilde{s}))$. In a similar manner, we write $\texttt{predOf}(\tilde{\eta})$ to mean $\bigwedge_{\tilde{l} \in \tilde{L}} (\tilde{\eta}(\tilde{l}) \Leftrightarrow \texttt{predOf}(\tilde{l}))$. Observe $vars(\texttt{predOf}(\tilde{\sigma})) \subseteq S$ and $vars(\texttt{predOf}(\tilde{\eta})) \subseteq S \cup L$. We abuse notation and write $val_{\sigma}(\tilde{\sigma})$ (resp. $val_{\sigma,\eta}(\tilde{\eta})$) to mean that $val_{\sigma}(\texttt{predOf}(\tilde{\sigma}))$ (resp. $val_{\sigma,\eta}(\texttt{predOf}(\tilde{\eta}))$) 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 $\texttt{predOf}(\tilde{v}) = \pi'$. We let $val_{\sigma,m}(\tilde{m})$ mean there is a bijection $h : \{1, ...|m|\} \to \{1, ...|\tilde{m}|\}$ s.t. we can associate to each $(lc, \eta)_i$ in m an $(lc, \tilde{\eta})_{h(i)}$ in $\tilde{m}$ such that $val_{\sigma,\eta}(\tilde{\eta})$ for each $i : 1 \leq i \leq |m|$.

The *abstraction* of (σ, m) is $\alpha((\sigma, m)) = \{(\tilde{\sigma}, \tilde{m}) | val_{\sigma}(\tilde{\sigma}) \wedge val_{\sigma,m}(\tilde{m})\}$.

Lemma 10. *If $(\tilde{\sigma}, \tilde{m}) \in \alpha((\sigma, m))$ then for each $\pi \in \textbf{preds}(\textbf{exprs}(S \cup L))$ s.t. $atoms(\pi) \subseteq \Pi$, we have $\sum_{\left\{(lc,\eta) | val_{\sigma,(lc,\eta)}(\pi)\right\}} m((lc, \eta)) = \sum_{\left\{(lc,\tilde{\eta}) | val_{\tilde{\sigma},(lc,\tilde{\eta})}(\pi)\right\}} \tilde{m}((lc, \tilde{\eta}))$.*

Proof. By definition of $\alpha((\sigma, m))$, $val_{\sigma,m}(\tilde{m})$ holds. This means that there is a bijection $h : \{1, ...|m|\} \to \{1, ...|\tilde{m}|\}$ s.t. we can associate to each $(lc, \eta)_i$ in m an $(lc, \tilde{\eta})_{h(i)}$ in $\tilde{m}$ such that $val_{\sigma,\eta}(\tilde{\eta})$ for each $i : 1 \leq i \leq |m|$. By construction of $val_{\sigma,\eta}(\tilde{\eta})$, we have that $val_{\tilde{\sigma},(lc,\tilde{\eta})}(\pi)$ and $val_{\sigma,(lc,\eta)}(\pi)$ coincide on each boolean combination of predicates in $\Pi_{mix} \cup \Pi_{loc}$. As a result, $val_{\tilde{\sigma},(lc,\tilde{\eta})}(\pi)$ and $val_{\sigma,(lc,\eta)}(\pi)$ coincide on each π such that $atoms(\pi) \subseteq \Pi$. This implies that $\sum_{\left\{(lc,\eta) | val_{\sigma,(lc,\eta)}(\pi)\right\}} m((lc, \eta)) = \sum_{\left\{(lc,\tilde{\eta}) | val_{\tilde{\sigma},(lc,\tilde{\eta})}(\pi)\right\}} \tilde{m}((lc, \tilde{\eta}))$ for each π with $atoms(\pi) \subseteq \Pi$.

□

Lemma 11. *Assume $(\tilde{\sigma}, \tilde{m}) \in \alpha((\sigma, m))$. If $val_{\sigma,m}(\omega_{inv})$ then the following holds: $val_{\tilde{m}}\left(\exists S.\textbf{predOf}(\tilde{\sigma}) \wedge \omega_{inv}\left[\left\{(\pi)^{\#} \leftarrow \sum_{\left\{(lc,\tilde{\eta}) | val_{(lc,\tilde{\eta})}(\pi)\right\}} c_{((lc,\tilde{\eta}))} | (\pi)^{\#} \in vars(\omega_{inv})\right\}\right]\right)$.*

Proof. We have that $val_{\sigma}(\texttt{predOf}(\tilde{\sigma}))$ holds since $(\tilde{\sigma}, \tilde{m}) \in \alpha((\sigma, m))$, and $val_{\sigma,m}(\omega_{inv})$ holds by assumption. Then, $val_{\sigma,m}(\texttt{predOf}(\tilde{\sigma}) \wedge \omega_{inv})$ also holds. By definition of how counted predicates are evaluated (see Sec. 4), the following also holds: $val_{\sigma}\left(\texttt{predOf}(\tilde{\sigma}) \wedge \omega_{inv}\left[\left\{(\pi)^{\#} \leftarrow \sum_{\left\{(lc,\eta) | val_{\sigma,(lc,\eta)}(\pi)\right\}} m((lc, \eta)) | (\pi)^{\#} \in vars(\omega_{inv})\right\}\right]\right)$. Since $atoms(\pi) \subseteq \Pi$ for each $(\pi)^{\#}$ appearing in $vars(\omega_{inv})$, Lem. 10 ensures: $val_{\sigma}\left(\texttt{predOf}(\tilde{\sigma}) \wedge \omega_{inv}\left[\left\{(\pi)^{\#} \leftarrow \sum_{\left\{(lc,\tilde{\eta}) | val_{(lc,\tilde{\eta})}(\pi)\right\}} \tilde{m}\tilde{\sigma}, (lc, \tilde{\eta}) | (\pi)^{\#} \in vars(\omega_{inv})\right\}\right]\right)$ holds. Finally, the existence of σ ensures that the following holds $\exists S.\texttt{predOf}(\tilde{\sigma}) \wedge \omega_{inv}\left[\left\{(\pi)^{\#} \leftarrow \sum_{\left\{(lc,\tilde{\eta}) | val_{(lc,\tilde{\eta})}(\pi)\right\}} \tilde{m}\tilde{\sigma}, (lc, \tilde{\eta}) | (\pi)^{\#} \in vars(\omega_{inv})\right\}\right]$. □

Lemma 3. *Any feasible P run has a feasible $\textbf{abstOf}_{\Pi}(P)$ run with a feasible run in any machine obtained as the strengthening of $enc(\textbf{abstOf}_{\Pi}(P))$ wrt. some P invariant $\omega_{inv} \in \textbf{preds}(\textbf{exprs}(S \cup \Omega_{Loc,S,L}))$.*

Proof. Proof of this lemma is by contradiction. Assume to the contrary that a P feasible run ρ_P exists but it does not have a $enc(\texttt{abstOf}_{\Pi}(P))_{str}$ feasible run $\rho_{enc(\texttt{abstOf}_{\Pi}(P))_{str}}$ where $enc(\texttt{abstOf}_{\Pi}(P))_{str}$ is a strengthening of machine $enc(\texttt{abstOf}_{\Pi}(P))$ with respect to an invariant ω_{inv}.

According to Def. 1, for each run ρ_P, a non-empty set $\alpha(\rho_P)$ of $\texttt{abstOf}_{\Pi}(P)$ feasible runs exist. Moreover, based on Lem. 1, for each run $\rho_{\texttt{abstOf}_{\Pi}(P)} \in \alpha(\rho_P)$ there exists an $enc(\texttt{abstOf}_{\Pi}(P))$ feasible run $\rho_{enc(\texttt{abstOf}_{\Pi}(P))}$ (before strengthening). So, if the run $\rho_{enc(\texttt{abstOf}_{\Pi}(P))_{str}}$ does not exist, it is because the run $\rho_{enc(\texttt{abstOf}_{\Pi}(P))}$ was not possible after the strengthening phase.

Let $\rho_{enc(\texttt{abstOf}_{\Pi}(P))} = (q_{\tilde{\sigma}_0}, \theta_{\tilde{m}_0}), \delta_1, \ldots (q_{\tilde{\sigma}_n}, \theta_{\tilde{m}_n})$ where $\rho_P = (\sigma_0, m_0), t_1, \ldots, (\sigma_n, m_n)$ with $(\tilde{\sigma}_i, \tilde{m}_i) \in \alpha((\sigma_i, m_i))$ for each $i : 0 \leq i \leq n$. Because $\rho_{enc(\texttt{abstOf}_{\Pi}(P))}$ is removed after strengthening, then there exists a step $(q_{\tilde{\sigma}}, \theta_{\tilde{m}}) \xrightarrow[enc(\texttt{abstOf}_{\Pi}(P))]{[q_{\tilde{\sigma}}:op:q_{\tilde{\sigma}'}]}$ $(q_{\tilde{\sigma}'}, \theta'_{\tilde{m}})$ in $\rho_{enc(\texttt{abstOf}_{\Pi}(P))}$ such that its corresponding step $(q_{\tilde{\sigma}}, \theta_{\tilde{m}}) \xrightarrow[\rho_{enc(\texttt{abstOf}_{\Pi}(P))_{str}}]{[q_{\tilde{\sigma}}:grd_{\tilde{\sigma}}(\omega_{inv});op;grd_{\tilde{\sigma}'}(\omega_{inv}):q_{\tilde{\sigma}'}]}$ $(q_{\tilde{\sigma}'}, \theta'_{\tilde{m}})$ is impossible. According to strengthening rule in Fig. 7, $grd_{\tilde{\sigma}}(\omega_{inv}) = \exists S.\texttt{predOf}(\tilde{\sigma}) \wedge \omega_{inv}\left[\left\{(\pi)^{\#} \leftarrow \sum_{\{(lc,\tilde{\eta})|val_{\tilde{\sigma},(lc,\tilde{\eta})}(\pi)\}} c_{(lc,\tilde{\eta})} | (\pi)^{\#} \in vars(\omega_{inv})\right\}\right]$. So, the fact that the mentioned step is not possible after strengthening implies that either (σ, θ_m) does not satisfy $grd_{\tilde{\sigma}}(\omega_{inv})$, or that $(\sigma', \theta_{m'})$ does not satisfy $grd_{\tilde{\sigma}'}(\omega_{inv})$. Both alternatives violate the fact that ω_{inv} is an invariant, that (σ, m) and (σ', m') are both reachable, and Lem. 11. □

Lemma 4. *State reachability is in general undecidable after strengthening.*

Proof. Sketch. We encode a two counters Minsky machine (Q, q_0, Δ, q_F) where Q is a finite set of states, q_0 is the initial state and q_F is the final one, and the transitions in Δ are either an increment $(q : (x_i)^{++} : q')$, a decrement $(q : x_i \geq 1; (x_i)^{--} : q')$ or a test for zero $(q : x_i = 0 : q')$ for one the counters $\{x_1, x_2\}$.

We construct a concurrent program with one main method and two processes $proc_1$ and $proc_2$ (see Fig. 17). The main method simulates the counter machine in the sense that a bijection associates each location $main@lc$ of the main procedure to a machine state in Q with lc_{ent} associated to q_0 and lc_{ext} associated to q_F. In addition, the main procedure has a transition $(lc_q \blacktriangleright lc_{q'} : stmt_{op})$ to each transition $(q : op : q')$ in Δ. More precisely,

1. if $op = (x_i)^{++}$, then $stmt_{op} = (count_i)^{++}; spawn(proc_i)$ (t_{inc} in Fig. 17);
2. if $op = (x_i)^{--}$, then $stmt_{op} = count_i \geq 1; (count_i)^{--}; join(proc_i)$ (t_{dec} in Fig. 17);
3. if $op = (x_i = 0)$, then $stmt_{op} = (count_i = 0)$ (t_{test} in Fig. 17)

Finally, we let $\omega_{trgt} = (main@lc_{ext})^{\#} \geq 1$ and $\omega_{inv} = (count_1 = (proc_1@lc_{ent})^{\#} + (proc_1@lc_{ext})^{\#}) \wedge (count_2 = (proc_2@lc_{ent})^{\#} + (proc_2@lc_{ext})^{\#})$ and $\Pi_0 = \{count_1 = 0, count_2 = 0\}$.

Predicate abstraction will maintain the predicates $count_i = 0$ but will loose their connection with the number of $proc_i$ processes. Strengthning restablishes this connection and we obtain, after strengthening, essentially the same counter machine as (Q, q_0, Δ, q_F) where the reachability of q_F is equivalent to the reachabilty if q_{trgt}. □

```
int count_1 := 0
int count_2 := 0

main :
  t_inc :  lc_q ▶ lc_q' :  (count_1)^{++}; spawn(proc_1)
  t_dec :  lc_q ▶ lc_q' :  count_2 ≥ 1; (count_2)^{--}; join(proc_2)
  t_test : lc_q ▶ lc_q' :  count_1 = 0
           ...

proc_1 :
  lc_ent ▶ lc_ext :  tt;

proc_1 :
  lc_ent ▶ lc_ext :  tt;
```

Fig. 17. Encoding a two counter machine.

Lemma 5. *For any configurations* (σ, m) *and* $(\tilde{\sigma}, \tilde{m})$ *s.t.* $(\tilde{\sigma}, \tilde{m}) \in \alpha((\sigma, m))$, *we have that* $\omega_{trgt}\left[\left\{(\pi)^{\#} \leftarrow \sum_{\{(lc,\eta)|val_{\sigma,(lc,\eta)}(\pi)\}} m((lc,\eta))|(\pi)^{\#} \in vars(\omega_{trgt})\right\}\right]$ *iff* $\omega_{trgt}\left[\left\{(\pi)^{\#} \leftarrow \sum_{\{(lc,\tilde{\eta})|val_{\tilde{\sigma},(lc,\tilde{\eta})}(\pi)\}} \tilde{m}((lc,\tilde{\eta}))|(\pi)^{\#} \in vars(\omega_{trgt})\right\}\right]$.

Proof. Recall $\omega_{trgt} \in \texttt{preds}(\texttt{exprs}(\Omega_{Loc,S,L}))$, and that $atoms(\pi) \subset \Pi$ for each $(\pi)^{\#}$ appearing in ω_{trgt}. Lem. 10 ensures that $\sum_{\{(lc,\eta)|val_{\sigma,(lc,\eta)}(\pi)\}} m((lc,\eta)) = \sum_{\{(lc,\tilde{\eta})|val_{(lc,\tilde{\eta})}(\pi)\}} \tilde{m}((lc,\tilde{\eta}))$ for each π such that $(\pi)^{\#}$ appears in $vars(\omega_{trgt})$. □

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 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.*

Proof. If a configuration satisfying ω_{trgt} is reachable in P then there is a P feasible run ρ_P that contains a configuration (σ, m) with $val_{\sigma,m}(\omega_{trgt})$ holds. Def. 1 ensures that $\alpha(\rho_P)$ is feasible. Let $(\tilde{\sigma}, \tilde{m}) \in \alpha((\sigma, m))$ appearing in $\alpha(\rho_P)$. Lem. 5 ensures that $\omega_{trgt}\left[\left\{(\pi)^{\#} \leftarrow \sum_{\{(lc,\tilde{\eta})|val_{\tilde{\sigma},(lc,\tilde{\eta})}(\pi)\}} \tilde{m}((lc,\tilde{\eta}))|(\pi)^{\#} \in vars(\omega_{trgt})\right\}\right]$ holds. This means q_{trgt} is reachable in $enc(\texttt{abstOf}_{\Pi}(P))$ (and Lem. 1) and this run is preserved after strengthening (Lem. 3). Since constrained monotonic abstraction is sound (Lem. 5), it cannot return `not_reachable`. In addition, if a P run ρ_P is returned by the simulation step, then it is possible in P. Furthermore, ρ_P was obtained from a run $\rho_{enc(\texttt{abstOf}_{\Pi}(P))}$ of $enc(\texttt{abstOf}_{\Pi}(P))$ that reached q_{trgt}. This means (Lem. 1) that a configuration $(\tilde{\sigma}, \tilde{m})$ in $\rho_{\texttt{abstOf}_{\Pi}(P)}$ (corresponding to a configuration (σ, m) in ρ_P with $(\tilde{\sigma}, \tilde{m}) \in \alpha((\sigma, m))$) satisfies $\omega_{trgt}\left[\left\{(\pi)^{\#} \leftarrow \sum_{\{(lc,\tilde{\eta})|val_{\tilde{\sigma},(lc,\tilde{\eta})}(\pi)\}} \tilde{m}((lc,\tilde{\eta}))|(\pi)^{\#} \in vars(\omega_{trgt})\right\}\right]$. Lem.

5 ensures that $\omega_{trgt}\left[\left\{(\pi)^{\#} \leftarrow \sum_{\left\{(lc,\eta)|val_{\sigma,(lc,\eta)}(\pi)\right\}} m((lc,\eta))|(\pi)^{\#} \in vars(\omega_{trgt})\right\}\right]$. Termination of each iteration of the outer loop is guaranteed by the effectiveness of the predicate abstraction and by the finiteness of the trace returned by constrained monotonic abstraction. Termination of each iteration of the inner loop is guaranteed by well quasi ordering (see [4]). □