Counting dynamically synchronizing processes

Zeinab Ganjei, Ahmed Rezine, Petru Eles and Zebo Peng

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Counting Dynamically Synchronizing Processes

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

Linköping University, Sweden

Received: date / Revised version: date

Abstract. We address the problem of automatically establishing correctness for programs generating an arbitrary number of concurrent processes and manipulating variables ranging over an infinite domain. The programs we consider can make use of the shared variables to count and synchronize the spawned processes. This allows them to implement intricate synchronization mechanisms, such as barriers. Automatically verifying correctness, and deadlock freedom, of such programs is beyond the capabilities of current techniques. For this purpose, we make use of *counting predicates* that mix counters referring to the number of processes satisfying certain properties and variables directly manipulated by the concurrent processes. We then combine existing works on counter, predicate, and constrained monotonic abstraction and build a nested counter example based refinement scheme for establishing correctness (expressed as non reachability of configurations satisfying counting predicates formulas.)We have implemented a tool (PACMAN, for predicated constrained monotonic abstraction) and used it to perform parameterized verification on several programs whose correctness crucially depends on precisely capturing the number of processes synchronizing using shared variables.

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

1 Introduction

We focus on automatically establishing synchronization related parameterized correctness. For this, we consider programs spawning arbitrarily many concurrent processes that use barriers or integer shared variables for counting the number of processes at different stages of their computations. Correctness is stated in terms of Safety properties expressed using *counting predicates*. Counting predicates make it possible to express statements about program variables and counters for the number of processes satisfying some predicates on program variables. Such statements can capture both individual properties, such as assertion violations, and global properties, such as deadlocks or relations between the numbers of processes at certain states.

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, are spawned in order to perform heavy computations in phases. For this reason, when not implemented directly using shared variables, constructs such as (dynamic) barriers are made available in mainstream libraries and programming languages such as java.util.concurrent, java.util.concurrent.Phaser, X10 or OpenMP.

Automatically taking into account the different phases by which arbitrary many processes can pass is beyond the capabilities of current automatic verification techniques. Indeed, and as an example, handling programs with barriers of arbitrary sizes (i.e. the number of processes participating in the barrier is fixed but arbitrarily large), is a non trivial task even in the case where all processes only manipulate boolean variables. In order to enforce the correct behaviour of a barrier, a verification procedure needs to capture relations between the number of processes satisfying certain properties, for instance that all processes are waiting at the barrier before any of them is allowed to cross it. This amounts to testing that the number of processes at certain locations is

* In part supported by the 12.04 CENIIT project.

zero. Checking violations of program assertions is then tantamount to checking state reachability of a counter machine where counters track the number of processes satisfying predicates such as being at some program location. No sound verification technique can therefore be complete for such systems.

Our approach to get around this problem builds on the following observation. In case there are no tests for the number of processes satisfying certain properties (e.g. being in specific programs locations for barriers), symmetric boolean concurrent programs can be exactly encoded as monotonic counter machines, i.e., essentially vector addition systems (VASs). For such systems, state reachability can be decided using a backwards exploration that only manipulates sets that are upward closed with respect to the component wise ordering [4, 14]. The approach is exact because of monotonicity of the induced transition system (more processes can fire more transitions since there are no tests on the numbers of processes). Termination is guaranteed by well quasi ordering of the component wise ordering on the natural numbers. The induced transition system is no more monotonic in the presence of tests on the number of processes. The idea in monotonic abstraction [6] is to modify the semantics of the entailed tests (e.g., zero tests for barriers), such that processes not satisfying the tests are removed (e.g., zero tests are replaced by resets). This results in a monotonic over-approximation of the original transition system and spurious traces are possible. This is also true for verification approaches that generate concurrent boolean programs with broadcasts as abstractions of concurrent programs manipulating integer variables (e.g., [11]). Such boolean approximations are monotonic even when the original program (before abstraction) can encode tests on the number of processes and has therefore a non monotonic invariant. Indeed, having more processes while respecting important relations between their numbers and certain variables in the original programs does not necessarily allow to fire more transitions (which is what abstracted programs do in such approaches).

To sum up, our approach consists in combining two nested counter example guided abstraction refinement loops. Each loop operates at a different level of abstraction. We summarize our contributions in the following points.

1. We introduce and propose to use *counting predicates* to express statements about program variables and about the number of processes satisfying given predicates on program variables.
2. We implement the outer loop by leveraging on existing symmetric predicate abstraction techniques. We encode resulting boolean programs in terms of counter machines where reachability of configurations satisfying a *counting predicates formula* is captured as a state reachability problem.
3. We explain how to strengthen the counter machine using *counting invariants*, i.e. counting predicates that hold on all runs. In this work, we automatically generate these invariants using classical thread modular analysis techniques.
4. We leverage on existing constrained monotonic abstraction techniques in order to implement the inner loop and to address the counter machine state reachability problem.
5. We have implemented both loops, together with automatic counting invariants generation, in a prototype (Pacman) that allowed us to automatically establish or refute counting predicate formulas such as deadlock freedom and assertions. All programs we report on may generate arbitrary many processes.
6. We include all proofs and make use of several examples to clarify the contributions.

The present article is an extended version of a previous conference paper [15].

Related work. Several works consider parameterized verification for concurrent programs. See [20] for a good survey. We report on some relevant recent techniques. The works in [17, 2] explore finite instances and automatically check for cutoff conditions. Except for checking larger instances, it is unclear how to refine entailed abstractions. [7] strengthens the approach of [2], but can not capture global properties that involve relations between number of processes and program variables, as we do in this work. In [12], the authors target verification of Petri nets which are inherently monotonic and generate invariants by weakening SMT formulas. Although we also target coverability, we do it for counter machines obtained by strengthening monotonic systems into non-monotonic ones. It is unclear how to apply this approach to our systems. The work in [19] gives a generalization of the IC3 algorithm and tries to build inductive invariants for well-structured transition systems. It is unclear how to adapt it to the kind of non-monotonic systems that we work with. Similar to [1], we combine auxiliary invariants obtained on certain variables in order to strengthen a reachability analysis. In [13], 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 control flow net. Such nets can model arbitraily many processes sharing global variables. We can also handle local variables and automatically discover relevant predicates by nesting symmetric predicate abstraction loop with a constrained monotonic abstraction loop. In [9], the authors present a highly optimized coverability checking approach for VASs with broadcasts. We need more than coverability of monotonic systems. In [18], the authors adopt symbolic representations that can track inter-thread predicates. This yields a non monotonic system and the authors force

```
    int wait, count := 0, 1;
    int cross, read:= 0, 0;

    process :
t0. pc0 → pc0 : spawn
t1. pc0 → pc1 : cross= 0; count := count + 1
t2. pc1 → pc2 : read := 1
// do some reading before the barrier
t3. pc2 → pc3 : read := 0
t4. pc3 → pc4 : wait := wait + 1
t5. pc4 → pc5 : (wait = count); cross := 1
//assert(read = 0)
t6. ...
```

Fig. 1. No matter how many processes are spawned, no process can be at $\mathtt{pc}_5$ and witness $\mathtt{read} > 0$.

monotonicity as in [6,5]. They however do not explain how to refine the obtained decidable monotonic abstraction for an undecidable problem. In [8], 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 closest works to ours are [5,11]. We introduced (constrained) monotonic abstraction in [6,5]. Monotonic abstraction was not combined with predicate abstraction, nor did it explicitly target counting properties or dynamic barrier based synchronization. In [11], the authors propose a predicate abstraction framework for concurrent multithreaded programs. As explained earlier, such abstractions cannot exclude behaviours forbidden by synchronization mechanisms such as barriers. In our work, we build on [11] 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 predicates in Sec. 4. Next, we explain the different phases of our nested loop in Sec. 5 and report on our experimental results in Sec. 6. We finally conclude in Sec. 7.

2 A Motivating Example

Consider the concurrent program listed in Fig. 1. In this example, processes share four integer variables, namely `wait`, `count`, `read` and `cross`. Variable `count` is initialized to 1, and variables `wait`,`cross` and `read` are initialized to 0. A single process starts executing the program. Arbitrarily many processes get spawned at location $\mathtt{pc}_0$ by transition $\mathtt{t}_0$.

Each process executes $\mathtt{t}_1$ and increments `count` if `cross` is 0, meaning that no process has crossed the barrier. Otherwise, no process can execute the transition $\mathtt{t}_1$. It then sets and rests `read`. Intuitively, `read` is greater than zero when there is at least a process doing some reading before the barrier between transitions $\mathtt{t}_2$ and $\mathtt{t}_3$. Transitions $\mathtt{t}_4$ and $\mathtt{t}_5$ essentially implement a barrier in the sense that all processes must have reached $\mathtt{pc}_4$ in order for any of them to move to location $\mathtt{pc}_5$. The first process that crosses the barrier changes `cross` from 0 to 1. As a result, no other process can take transition $\mathtt{t}_1$ and start working. After the barrier, no process should be left behind. We capture this by asserting that no process at location $\mathtt{pc}_5$ should witness $\mathtt{read} > 0$. We write $@\mathtt{pc}_5$ to mean the predicate satisfied by all processes at location $\mathtt{pc}_5$. A process that satisfies $@\mathtt{pc}_5 \wedge (\mathtt{read} > 0)$ is at location $\mathtt{pc}_5$ and witnesses $\mathtt{read} > 0$. A configuration of the program satisfies the predicate $(@\mathtt{pc}_5 \wedge (\mathtt{read} > 0))^{\#} \geq 1$ if the number of such processes is greater than or equal to one. We call such a predicate a *counting predicate* (introduced in Sec 4). Counting predicates can be used to capture configurations violating other properties than assertions, e.g., deadlock freedom.

The assertion $(@\mathtt{pc}_5 \wedge (\mathtt{read} > 0))^{\#} \geq 1$ is never violated under any run starting from a configuration in which a single process starts executing from location $\mathtt{pc}_0$. In order to establish this fact, any verification procedure needs to take into account the barrier at $\mathtt{t}_5$ as well as the two sources of infinitness; namely, the infinite domain of the shared and local variables and the number of processes that may participate in the run. Apart from [13] that cannot handle local variables, the closest works to ours deal with these two sources of infinitness separately and cannot capture facts that relate them, namely, the values of the program variables and the number of generated processes.

Any sound analysis that does not take into account that the `count` variable captures the number of processes at locations $\mathtt{pc}_1$ or later, and that `wait` represents the number of processes at locations $\mathtt{pc}_4$ or later, will not be able to discard scenarios where a process executes $\mathtt{read} := 1$ although one of them is at $\mathtt{pc}_5$. Such an analysis will therefore fail to show that $\mathtt{read} = 0$ each time a process is at $\mathtt{pc}_5$.

Our tool, 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 which consists of two nested CEGAR loops in the remaining on the example of Fig. 1.

From concurrent programs to boolean concurrent programs. We build on the recent predicate abstraction techniques for concurrent programs. Such techniques first

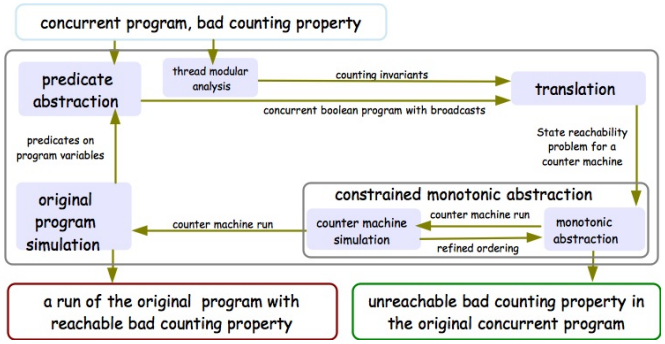

Fig. 2. **P**redic**a**ted **C**onstrained **M**onotonic **A**bstractio**n**

discard all variables and predicates and only keep the control flow. This leads to a number of counter example guided abstraction refinement steps (the outer CEGAR loop in Fig. 2) that will result in the addition of new predicates. Our implementation automatically adds the predicates `cross_leq_0`, `read_leq_0`, `wait_leq_count` and `count_leq_wait`.

It is worth noticing that all variables of the obtained concurrent program are finite state (in fact boolean). Hence, one would need a finite number of counters in order to faithfully capture the behaviour of the abstracted program using counter abstraction.

From concurrent boolean programs to counter machines. Given a concurrent boolean program and a property to be checked, we generate a counter machine that essentially boils down to a vector addition system with transfers (with additional tests for global properties such as deadlock freedom). Each counter in the machine counts the number of processes at some location with some specific value combination for the local variables. One state in the counter machine represents reaching a configuration violating the property we want to check. The other states correspond to the possible combinations of the global variables. Such a machine cannot relate the number of processes in certain locations to the predicates that are valid at certain states (for instance that $\texttt{count} = \texttt{wait}$). These are essential for verification of programs where counters are used to synchronize processes. In order to remedy to this fact, we make use of *counting invariants* that relate program variables, `count` and `wait` in the following invariants, to the number of processes at certain locations.

$$\texttt{count} = \sum_{\texttt{i}\geq 1}(@\texttt{pc}_\texttt{i})^{\#}$$

$$\texttt{wait} = \sum_{\texttt{i}\geq 4}(@\texttt{pc}_\texttt{i})^{\#}$$

We automatically generate such invariants using a simple thread modular analysis that tracks the number of processes satisfying some property.

Example 1 (Thread modular analysis). To recover the two counting invariants above, we can perform a classical thread modular analysis [16] where we add a shared instrumentation variable $\_\texttt{pc}_i$ to track the number of processes at location $\texttt{pc}_i$ for $i : 0 \leq i \leq 6$. We use a suitable abstract numerical domain (in this case polyhedral domain). For the program of Fig. 1, we obtained the counting invariants mentioned earlier as well as other invariants such as $0 \leq \texttt{count}, 0 \leq \texttt{wait}$ and $\texttt{wait} \leq \texttt{count}$ that helped for pruning the state space as mentioned in Sec. 6.

Given such counting invariants, we constrain the counter machine and generate a more precise machine that may not be a vector addition system anymore. We explain in Sect. 5 that the resulting state reachability problem is now undecidable in general.

Constrained monotonic abstraction. We monotonically abstract the resulting counter machine in order to answer the state reachability problem. Spurious traces are now possible. Essentially, monotonic abstraction closes upwards the obtained sets of predecessor configurations. This over-approximation might add larger configurations that did not belong to the set of predecessor configurations. Intuitively, the effect of monotonic abstraction "in forward" on the example of Fig.1 is that it "deletes" processes violating the constraint imposed by the barrier [6]. This example illustrates a situation where such approximations yield false positives. To see this, suppose two processes exist. A first process gets to $\texttt{pc}_4$ and waits. The second process moves to $\texttt{pc}_2$. Deleting the second process, is allowed by the monotonic abstraction and opens the barrier for the first process. However, the assertion can now be violated because the deleted process did not have time to reset the variable `read`. 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 $\texttt{pc}_1$ is zero, then closing upwards will not alter this fact. By doing so, the process that was deleted in forward at $\texttt{pc}_2$ is not allowed to be there to start with, and the assertion is automatically established for any number of processes. The inner loop of our approach can automatically perform more elaborate refinements such as comparing the number of processes at different locations. Exact traces of the counter machine are sent to the next step and unreachability of the control location establishes safety of the concurrent program.

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

3 Preliminaries

We write $\mathbb{N}$ and $\mathbb{Z}$ to mean the sets of natrual and integer values respectively. Given two natural numbers $i, j \in \mathbb{N}$, we use $[i, j]$ to denote the set $\{k \in \mathbb{N} | i \leq k \leq j\}$. We let $\mathbb{B} = \{\mathtt{tt}, \mathtt{ff}\}$ be the set of boolean values. In this section, we write V and V^b to respectively mean a set of integer and boolean variables. We write X to mean some set V or V^b. In a similar manner, we write $\mathtt{v}$ and $\mathtt{v}^b$ to respectively mean an integer or a boolean variable. We also write x to mean a variable of some type.

We write $\mathtt{exprsOf}(V)$ to mean the set of arithmetic expressions over the integer variables V. An arithmetic expression $\mathtt{e}$ (or expression for short) in $\mathtt{exprsOf}(V)$ is an integer constant k, an integer variable $\mathtt{v}$, or the sum or difference of two expressions as described below:

$$\mathtt{e} ::= \mathtt{k} \mid \mathtt{v} \mid (\mathtt{e} + \mathtt{e}) \mid (\mathtt{e} - \mathtt{e}) \mid \mathtt{k}\,\mathtt{e} \qquad \mathtt{v} \in V$$

We let $\sim$ be a comparator in $\{<, \leq, =, \geq, >\}$. We write $\mathtt{predsOf}_E^{V^b}$ to mean the set of predicates (i.e., boolean expressions) over boolean variables V^b and arithmetic expressions E. A predicate π in $\mathtt{predsOf}_E^{V^b}$ is either a boolean value $\mathtt{b}$, a variable $\mathtt{v}^b$ in V^b, a comparison of two expressions in E or a boolean combination of predicates. It takes the following form:

$$\pi ::= \mathtt{b} \mid \mathtt{v}^b \mid (\mathtt{e} \sim \mathtt{e}) \mid \neg\pi \mid \pi \wedge \pi \mid \pi \vee \pi \quad \mathtt{v}^b \in V^b, \mathtt{e} \in E$$

We write $vars(\mathtt{e})$ to mean all integer variables appearing in an expression $\mathtt{e}$, and $vars(\pi)$ to mean all variables appearing in π, namely both boolean variables appearing in π and all integer variables in $vars(\mathtt{e})$ for each $\mathtt{e}$ appearing in π. We also write $\mathtt{comparisonsOf}(\pi)$ to mean all comparisons $(\mathtt{e} \sim \mathtt{e})$ appearing in π. We assume in the following an arithmetic or boolean expression $\mathtt{exp}$ or any indexed version of it. A mapping $\mathbb{x} : X \to Y$ associates an expression to each variable in X. Expressions in Y have the same type as the variables in X. We often write a mapping $\mathbb{x} : X \to Y$ as the set $\{x \leftarrow \mathbb{x}(x) | x \in X\}$. We write $\mathtt{exp}[\mathbb{x}]$ to mean the evaluation of expression $\mathtt{exp}$ with respect to a mapping $\mathbb{x}$. We perform the evaluation as follows. First, an expression $\mathtt{exp}_{tmp}$ is deduced by syntactically and simultaneously replacing in $\mathtt{exp}$ each occurrence of a variable $x \in X$ by the corresponding $\mathbb{x}(x)$. Then, if $vars(\mathtt{exp}_{tmp}) = \emptyset$, $\mathtt{exp}[\mathbb{x}]$ is the constant obtained by evaluating $\mathtt{exp}_{tmp}$. Otherwise, $\mathtt{exp}[\mathbb{x}]$ is taken to be $\mathtt{exp}_{tmp}$ (see Ex. 2). Let $\mathbb{x} : X \to Y$ and $\mathbb{x}' : X' \to Y'$ be two mappings If X and X' are disjoint, we write $\mathtt{exp}[\mathbb{x}, \mathbb{x}']$ to mean the evaluation of expression $\mathtt{exp}$ with respect to $\mathbb{x} \cup \mathbb{x}'$. Larger unions of mappings with pairwise disjoint domains are handled in a similar manner. We abuse notation and write $\mathbb{x}[\mathbb{x}']$ to mean $\{x \leftarrow \mathbb{x}(x) | x \in X \setminus X'\} \cup \{x \leftarrow \mathbb{x}'(x) | x \in X \cap X'\}$.

$$\begin{aligned}
\mathtt{prog} ::= {} & (\mathtt{s} := (k \mid \star))^* \\
& \mathtt{process}: \\
& (\mathtt{l} := (k \mid \star))^* \\
& (\mathtt{pc} \to \mathtt{pc} : \mathtt{stmt})^+ \\
\mathtt{stmt} ::= {} & \mathtt{v}_1, \ldots, \mathtt{v}_n := \mathtt{e}_1, \ldots, \mathtt{e}_n \\
& \mid \mathtt{spawn} \mid \mathtt{join} \mid \pi \mid \mathtt{stmt}; \mathtt{stmt}
\end{aligned}$$

Fig. 3. Syntax of concurrent programs: $\mathtt{s}$ is a shared variable in S, $\mathtt{l}$ is a local variable in L, $\mathtt{v}_1, \ldots \mathtt{v}_n$ are pairwise different variables in $S \cup L$, $\mathtt{e}_1, \ldots \mathtt{e}_n$ are arithmetic expressions in $\mathtt{exprsOf}(S \cup L)$ and π is a predicate in $\mathtt{predsOf}_{\mathtt{exprsOf}(S \cup L)}$.

Example 2 (Expressions). Let $\pi = \mathtt{v}^b \wedge (\mathtt{v} = \mathtt{v}' + 1)$. Then, $vars(\pi) = \{\mathtt{v}^b, \mathtt{v}, \mathtt{v}'\}$. If we let $\mathbb{x}^b = \{\mathtt{v}^b \leftarrow \mathtt{tt}\}$, $\mathbb{x}_1 = \{\mathtt{v} \leftarrow \mathtt{v}' + 5, \mathtt{v}' \leftarrow 3\}$ and $\mathbb{x}_2 = \{\mathtt{v}' \leftarrow -1\}$, then $\pi[\mathbb{x}^b, \mathbb{x}_1] = \mathtt{tt} \wedge (\mathtt{v}' + 5 = 3 + 1)$ and $\pi[\mathbb{x}^b, \mathbb{x}_1][\mathbb{x}_2] = \mathtt{tt}$.

A multiset $\mathbb{m}$ over a set Σ is a mapping $\Sigma \to \mathbb{N}$. We sometimes write a multiset $\mathbb{m}$ by enumerating its elements in some predefined total order on Σ, i.e. as $[\sigma, \ldots]$ where each $\sigma \in \Sigma$ appears exactly $\mathbb{m}(\sigma)$ times. A bijection from a multiset $[\sigma_1, \sigma_2, \ldots \sigma_n]$ to a multiset $[\sigma'_1, \sigma'_2, \ldots, \sigma'_n]$ of the same size is a bijection from $[1, n]$ to $[1, n]$ that associates each element of the former multiset to an element of the latter. The size $|\mathbb{m}|$ of a multiset $\mathbb{m}$ is $\sum_{\sigma \in \Sigma} \mathbb{m}(\sigma)$. We write $\sigma \oplus \mathbb{m}$ to mean the multiset $\mathbb{m}'$ such that $\mathbb{m}'(\sigma')$ equals $\mathbb{m}(\sigma) + 1$ if $\sigma = \sigma'$ and $\mathbb{m}(\sigma)$ otherwise.

Example 3 (Multisets). Let $\Sigma = \{a, b, c\}$, and define $\mathbb{m} = [a, a]$ and $\mathbb{m}' = [c, c]$. A possible bijection from multiset $b \oplus \mathbb{m} = [a, a, b]$ to multiset $b \oplus \mathbb{m}' = [b, c, c]$ is the mapping $\{1 \leftarrow 2, 2 \leftarrow 1, 3 \leftarrow 3\}$ that sends the first a to the first c, the second a to b and b to the second c.

4 Concurrent Programs and Counting Predicates.

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. The procedure of a program $P = (S, L, T)$ is given in terms of a finite set T of transitions, each of the form $(\mathtt{pc}_1 \to \mathtt{pc}'_1 : \mathtt{stmt}_1)$. Transitions operate on two finite sets of integer variables, namely a set S of shared variables and a set L of local variables. Each transition $(\mathtt{pc} \to \mathtt{pc}' : \mathtt{stmt})$ involves two program locations $\mathtt{pc}$ and $\mathtt{pc}'$ and a statement $\mathtt{stmt}$. We write PC to mean the set of all locations appearing in T. We always distinguish two locations, namely an entry location $\mathtt{pc}_0$ and an exit location $\mathtt{pc}_x$. Program syntax is described in Fig. 3.

$$\frac{(\mathbb{s},\mathbb{l},\mathbb{m}) \overset{\texttt{stmt}}{\underset{P}{\succ\!\!\rightarrow}} (\mathbb{s}',\mathbb{l}',\mathbb{m}')}{(\mathbb{s},(\texttt{pc},\mathbb{l})\oplus\mathbb{m}) \overset{(\texttt{pc}\to\texttt{pc}':\texttt{stmt})}{\underset{P}{\longrightarrow}} (\mathbb{s}',(\texttt{pc}',\mathbb{l}')\oplus\mathbb{m}')} : transition$$

$$\frac{(\mathbb{s},\mathbb{l},\mathbb{m}) \overset{\texttt{stmt}}{\underset{P}{\succ\!\!\rightarrow}} (\mathbb{s}',\mathbb{l}',\mathbb{m}') \text{ and } (\mathbb{s}',\mathbb{l}',\mathbb{m}') \overset{\texttt{stmt}'}{\underset{P}{\succ\!\!\rightarrow}} (\mathbb{s}'',\mathbb{l}'',\mathbb{m}'')}{(\mathbb{s},\mathbb{l},\mathbb{m}) \overset{\texttt{stmt;stmt}'}{\underset{P}{\succ\!\!\rightarrow}} (\mathbb{s}'',\mathbb{l}'',\mathbb{m}'')} : sequence$$

$$\frac{\mathbb{v} = \{\mathbb{v}_i \leftarrow \mathbb{e}_i[\mathbb{s},\mathbb{l}] | i : 1 \le i \le n\}}{(\mathbb{s},\mathbb{l},\mathbb{m}) \overset{v_1,\ldots v_n,:=\mathbb{e}_1,\ldots\mathbb{e}_n}{\underset{P}{\succ\!\!\rightarrow}} (\mathbb{s}[\mathbb{v}],\mathbb{l}[\mathbb{v}],\mathbb{m})} : assign$$

$$\frac{\pi[\mathbb{s},\mathbb{l}]}{(\mathbb{s},\mathbb{l},\mathbb{m}) \overset{\pi}{\underset{P}{\succ\!\!\rightarrow}} (\mathbb{s},\mathbb{l},\mathbb{m})} : assume$$

$$\frac{\mathbb{m}' = (\texttt{pc}_0,\mathbb{l}_{init})\oplus\mathbb{m} \text{ with } \mathbb{l}_{init}\in\mathbb{L}_{init}}{(\mathbb{s},\mathbb{l},\mathbb{m}) \overset{\texttt{spawn}}{\underset{P}{\succ\!\!\rightarrow}} (\mathbb{s},\mathbb{l},\mathbb{m}')} : spawn$$

$$\frac{\mathbb{m} = \left((\texttt{pc}_x,\mathbb{l}')\oplus\mathbb{m}'\right)}{(\mathbb{s},\mathbb{l},\mathbb{m}) \overset{\texttt{join}}{\underset{P}{\succ\!\!\rightarrow}} (\mathbb{s},\mathbb{l},\mathbb{m}')} : join$$

Fig. 4. Semantics of concurrent programs. Executions start from some $(\texttt{pc}_0,\mathbb{m}_{init})$ with $\mathbb{m}_{init}\in\mathbb{M}_{init}$.

Semantics. Initially, a single process starts executing the procedure with both local and shared variables initialized as stated in their respective declarations. Executions might involve an arbitrary number of spawned processes. The execution of any process (whether initial or spawned with a `spawn` statement) starts at the entry location $\texttt{pc}_0$ with the corresponding local variables initialized as stated in their respective declarations. Any process at an exit point $\texttt{pc}_x$ 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 by a single process without interruption from other processes.

More formally, a configuration is given in terms of a shared state and a processes configuration. A *shared state* $\mathbb{s} : S \to \mathbb{Z}$ is a mapping that associates an integer to each variable in S. We write $\mathbb{S}$ to mean the set of all shared states. An *initial shared state* is a mapping in $\mathbb{S}$ that respects shared variables declarations. We write $\mathbb{S}_{init}$ to mean the set of all initial shared states. A *process state* is a pair $(\texttt{pc},\mathbb{l})$ where the location pc belongs to PC and the *local state* $\mathbb{l} : L \to \mathbb{Z}$ maps each local variable to an integer number. We also write $\mathbb{L}$ and $\mathbb{L}_{init}$ to respectively mean the sets of all local states and all initial local states. A *processes configuration* is a multiset $\mathbb{m}$ over process states. An *initial processes configuration* maps all $(\texttt{pc},\mathbb{l})$ to 0 except for a single $(\texttt{pc}_0,\mathbb{l})$, with $\mathbb{l}\in\mathbb{L}_{init}$, mapped to 1. We write $\mathbb{M}$ and $\mathbb{M}_{init}$ to mean the sets of all processes configurations and initial processes configurations respectively. Finally, a *configuration* is a pair $(\mathbb{s},\mathbb{m})$ consisting of a shared state $\mathbb{s}$ and a processes configuration $\mathbb{m}$. We write $(\mathbb{s},\mathbb{m}) \overset{t}{\underset{P}{\rightarrow}} (\mathbb{s}',\mathbb{m}')$ to mean that the transition t of the form $(\texttt{pc}\to\texttt{pc}':\texttt{stmt})$ applies atomically to configuration $(\mathbb{s},\mathbb{m})$ and changes it to $(\mathbb{s}',\mathbb{m}')$. We introduce a relation $\overset{\texttt{stmt}}{\underset{P}{\succ\!\!\rightarrow}}$ in order to describe the steps involved in the semantics of transitions (Fig. 4). We write $(\mathbb{s},\mathbb{l},\mathbb{m}) \overset{\texttt{stmt}}{\underset{P}{\succ\!\!\rightarrow}} (\mathbb{s}',\mathbb{l}',\mathbb{m}')$, where $\mathbb{s},\mathbb{s}'$ are shared states, $\mathbb{l},\mathbb{l}'$ are local states, and $\mathbb{m},\mathbb{m}'$ are multisets of process configurations, in order to mean that a process of the program P at local state $\mathbb{l}$ when the shared state is $\mathbb{s}$ and the other process configurations are captured by $\mathbb{m}$, can execute the statement `stmt` and take the program to a configuration where the process has local state $\mathbb{l}'$, the shared state is $\mathbb{s}'$ and the configurations of the other processes are captured by $\mathbb{m}'$. For instance, a process can always execute a join if there is another process at location $\texttt{pc}_x$ (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*). A P *run* ρ is a configuration starting alternating sequence of transitions and configurations $(\mathbb{s}_0,\mathbb{m}_0)\ t_1\ldots t_n(\mathbb{s}_n,\mathbb{m}_n)$. The run is P *feasible* if $(\mathbb{s}_i,\mathbb{m}_i) \overset{t_{i+1}}{\underset{P}{\rightarrow}} (\mathbb{s}_{i+1},\mathbb{m}_{i+1})$ for each $i\ :\ 0\le i<n$ and $\mathbb{s}_0$ and $\mathbb{m}_0$ are initial, i.e., $\mathbb{s}_0\in\mathbb{S}_{init}$ and $\mathbb{m}_0\in\mathbb{M}_{init}$. Each configuration $(\mathbb{s}_i,\mathbb{m}_i)$ for $i\ :\ 0\le i\le n$ is then said to be *reachable*.

Example 4 (Feasible run). Consider the concurrent program in Fig. 1. An initial configuration is $(\mathbb{s}_0,\mathbb{m}_0)$ where $\mathbb{s}_0 = \{\texttt{count}\leftarrow 1,\texttt{wait}\leftarrow 0,\texttt{cross}\leftarrow 0,\texttt{read}\leftarrow 0\}$ and the initial multiset $\mathbb{m}_0$ associates 1 to the unique process state with location $\texttt{pc}_0$ and 0 to all other process states. A feasible run is then represented below.

$\mathbb{s}$				$\mathbb{m}$
count	wait	cross	read	pc
1	**0**	**0**	**0**	$\texttt{pc}_0$
$\Downarrow\ \texttt{t}_0.\texttt{pc}_0 \to \texttt{pc}_0$: spawn				
1	**0**	**0**	**0**	$\texttt{pc}_0,\texttt{pc}_0$
$\Downarrow\ \texttt{t}_0.\texttt{pc}_0 \to \texttt{pc}_0$: spawn				
1	**0**	**0**	**0**	$\texttt{pc}_0,\texttt{pc}_0,\texttt{pc}_0$
$\Downarrow\ \texttt{t}_1.\texttt{pc}_0 \to \texttt{pc}_1$: cross$=0$; count $:=$ count $+1$				
2	**0**	**0**	**0**	$\texttt{pc}_0,\texttt{pc}_0,\texttt{pc}_1$
$\Downarrow\ \texttt{t}_2.\texttt{pc}_1 \to \texttt{pc}_2$: read $:=1$				
2	**0**	**0**	**1**	$\texttt{pc}_0,\texttt{pc}_0,\texttt{pc}_2$
$\Downarrow\ \texttt{t}_3.\texttt{pc}_2 \to \texttt{pc}_3$: read $:=0$				
2	**0**	**0**	**0**	$\texttt{pc}_0,\texttt{pc}_0,\texttt{pc}_3$

Counting Predicates. Recall that PC is the set of program locations. We make use of a set of boolean variables $\{@\texttt{pc} \mid \texttt{pc}\in PC\}$, denoted $@PC$. Intuitively, a process evaluates @pc to `tt` exactly when it is at location pc. This way, we can build boolean expressions in the set $\texttt{predsOf}^{@PC}_{\texttt{exprsOf}(S\cup L)}$. With these predicates we can state facts about both the location of some process, as well as its own local variables and the values of the shared variables. For instance, at the fourth step of the run depicted

in Ex. 4, there is one process for which ($@\mathtt{pc}_2 \wedge \mathtt{count} \geq 1$) holds.

We associate a *counting variable* $(\pi)^{\#}$ to each predicate π in $\mathtt{predsOf}^{@PC}_{\mathtt{exprsOf}(S \cup L)}$. Intuitively, in a given program configuration, the variable $(\pi)^{\#}$ counts the number of processes for which the predicate π holds. We denote the counting variables $\left\{(\pi)^{\#} | \pi \in \mathtt{predsOf}^{@PC}_{\mathtt{exprsOf}(S \cup L)}\right\}$ with $\Omega_{PC,S,L}$. For example, $(@\mathtt{pc}_2 \wedge \mathtt{count} \geq 1)^{\#}$ is a variable that counts the number of processes for which $(@\mathtt{pc}_2 \wedge \mathtt{count} \geq 1)$ holds. Such counting a variable is evaluated with respect to a shared state $\mathbb{s}$ and a process configuration $(\mathtt{pc}, \mathbb{l})$. We abuse notation and write $\mathtt{v}[\mathbb{s}, (\mathtt{pc}, \mathbb{l})]$ to mean the variable $\mathtt{v}$ participating in a counting variable, is evaluated to $\mathbb{s}(\mathtt{v})$ if $\mathtt{v} \in S$, to $\mathbb{l}(\mathtt{v})$ if $\mathtt{v} \in L$, or to $(\mathtt{pc} = \mathtt{pc}')$ if v is the boolean variable $@\mathtt{pc}'$.

Any predicate in $\mathtt{predsOf}_{\mathtt{exprsOf}(S \cup \Omega_{PC,S,L})}$ is a *counting predicate*. We need a shared state $\mathbb{s}$ and a processes configuration $\mathbb{m}$ in order to evaluate a counting variable (Eq.1) or a counting predicate (Eq.2). We abuse notation and write $\omega[\mathbb{s}, \mathbb{m}]$ to mean the evaluation of the counting predicate ω wrt. a configuration $(\mathbb{s}, \mathbb{m})$. The evaluation is performed as follows. Given a configuration $(\mathbb{s}, \mathbb{m})$, a shared variable $\mathtt{s} \in S$ is evaluated as usual to $\mathtt{s}[\mathbb{s}, \mathbb{m}] = \mathbb{s}(\mathtt{s})$; whereas the counting variable $(\pi)^{\#}$ is evaluated to the number of processes satisfying π in $(\mathbb{s}, \mathbb{m})$.

$$(\pi)^{\#}[\mathbb{s}, \mathbb{m}] = \sum_{\{(\mathtt{pc}, \mathbb{l}) \text{ s.t. } \pi[\mathbb{s}, (\mathtt{pc}, \mathbb{l})]\}} \mathbb{m}((\mathtt{pc}, \mathbb{l})) \quad (1)$$

$$\omega[\mathbb{s}, \mathbb{m}] = \omega[\mathbb{s}, \{(\pi)^{\#} \leftarrow (\pi)^{\#}[\mathbb{s}, \mathbb{m}] | (\pi)^{\#} \in vars(\omega)\}] \quad (2)$$

Our counting predicates are quite expressive. For instance, we can capture assertion violations, deadlocks or rich program invariants with them (see Ex. 5). For any $\mathtt{pc}$, we can define a counting predicate $\mathtt{isEnabled}(\mathtt{pc})$ that captures whether a process currently at location $\mathtt{pc}$ can fire some transition. For instance, in the running example of Fig. 1, $\mathtt{isEnabled}(\mathtt{pc}_0) = \mathtt{true}$ and $\mathtt{isEnabled}(\mathtt{pc}_4) = (\mathtt{wait} = \mathtt{count})$. If there would have been only one transition from $\mathtt{pc}_6$ consisting in a join operation, then $\mathtt{isEnabled}(\mathtt{pc}_6)$ would have been $(@\mathtt{pc}_x \geq 1)^{\#}$.

Example 5 (Counting predicates). The following counting predicates capture configurations of the program of Fig. 1: ω_1 captures configurations that violate the assertion, ω_2 captures those where a deadlock occurs, and ω_3 an over-approximation of the reachable configurations (i.e., an invariant).

$$\omega_1 = (@\mathtt{pc}_5 \wedge (\mathtt{read} > 0))^{\#} \geq 1$$

$$\omega_2 = \textstyle\bigwedge_{\mathtt{pc} \in PC} (@\mathtt{pc} \wedge \mathtt{isEnabled}(\mathtt{pc}))^{\#} = 0$$

$$\omega_3 = (\mathtt{count} = \textstyle\sum_{i \geq 1} (@\mathtt{pc}_i)^{\#}) \wedge (\mathtt{wait} = \textstyle\sum_{i \geq 4} (@\mathtt{pc}_i)^{\#})$$

bool $\mathtt{read\_leq\_0}, \mathtt{wait\_leq\_count} := \mathtt{tt}, \mathtt{tt}$

process :

$\mathtt{t}^b_0$. $\mathtt{pc}_0 \to \mathtt{pc}_0$: $\mathtt{spawn}$

$\mathtt{t}^b_1$. $\mathtt{pc}_0 \to \mathtt{pc}_1$: $(\mathtt{tt}); \mathtt{wait\_leq\_count} := \mathtt{ch}(\mathtt{wait\_leq\_count}, \mathtt{ff})$

$\mathtt{t}^b_2$. $\mathtt{pc}_1 \to \mathtt{pc}_2$: $\mathtt{read\_leq\_0} := \mathtt{ff}$

$\mathtt{t}^b_3$. $\mathtt{pc}_2 \to \mathtt{pc}_3$: $\mathtt{read\_leq\_0} := \mathtt{tt}$

$\mathtt{t}^b_4$. $\mathtt{pc}_3 \to \mathtt{pc}_4$: $\mathtt{wait\_leq\_count} := \mathtt{ch}(\mathtt{ff}, \neg\mathtt{wait\_leq\_count})$

$\mathtt{t}^b_5$. $\mathtt{pc}_4 \to \mathtt{pc}_5$: $(\mathtt{wait\_leq\_count}); (\mathtt{tt})$

$\mathtt{t}^b_6$. ...

Fig. 5. Predicate abstraction of the program in Fig. 1 with respect to the predicates $\Pi = \{\mathtt{read\_leq\_0}, \mathtt{wait\_leq\_count}\}$.

5 Relating abstraction layers

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 predicate abstraction technique in [11] in order to generate an abstraction in the form of a boolean program $\mathtt{abst}_{\Pi}(P) = \left(S^b, L^b, T^b\right)$ where all shared and local variables are boolean. 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 $\mathtt{originOf}(\mathtt{v}^b)$ in Π_{shr} (resp. $\Pi_{mix} \cup \Pi_{loc}$) to each boolean variable $\mathtt{v}^b$ in S^b (resp. L^b). The function is a bijection as each predicate will be associated to one and only one boolean variable (that tracks the value of that predicate) and vice versa.

Example 6 (Predicate abstraction). Consider the concurrent program in Fig. 1. We implemented the predicate abstraction of [11] which results, for the predicates $\Pi = \{\mathtt{read\_leq\_0}, \mathtt{wait\_leq\_count}\}$, in the boolean program of Fig. 5. A bijection associates the predicates $\mathtt{wait} \leq \mathtt{count}$ and $\mathtt{read} \leq 0$ in Π_{shr} respectively to the boolean variables $\mathtt{wait\_leq\_count}$ and $\mathtt{read\_leq\_0}$ in S^b of the boolean program.

In addition, there are as many transitions in T as in T^b. For each $(\mathtt{pc} \to \mathtt{pc}' : \mathtt{stmt})$ in T there is a corresponding $(\mathtt{pc} \to \mathtt{pc}' : \mathtt{abst}_{\Pi}(\mathtt{stmt}))$ with the same source and destination locations $\mathtt{pc}, \mathtt{pc}'$, but with an abstracted statement $\mathtt{abst}_{\Pi}(\mathtt{stmt})$ that may operate on the variables $S^b \cup L^b$. Moreover, abstracted statements can mention the local variables of passive processes, i.e., processes other than the one executing the transition. For

$$
\begin{aligned}
\texttt{prog} ::= & \left(\texttt{s}^b := \left(\texttt{tt} \;\middle|\; \texttt{ff} \;\middle|\; *\right)\right)^* \\
& \texttt{process}: \\
& \left(\texttt{l}^b := \left(\texttt{tt} \;\middle|\; \texttt{ff} \;\middle|\; *\right)\right)^* \\
& \left(\texttt{pc} \rightarrow \texttt{pc} : \texttt{stmt}\right)^+ \\
\texttt{stmt} ::= & \texttt{v}_1^b, \ldots, \texttt{v}_n^b := \texttt{ch}\left(\pi_1, \pi_1'\right), \ldots, \texttt{ch}\left(\pi_n, \pi_n'\right) \\
& \texttt{spawn} \;\middle|\; \texttt{join} \;\middle|\; \pi \;\middle|\; \texttt{stmt}; \texttt{stmt}
\end{aligned}
$$

Fig. 6. Syntax of concurrent boolean programs.

this, we make use of the variables $L_p^b = \left\{\texttt{l}_p^b | \texttt{l}^b \text{ in } L^b\right\}$ where each $\texttt{l}_p^b$ denotes local variable $\texttt{l}^b$ of passive processes. We use passive variables to capture broadcasts where local variables of all passive processes need to be updated. Note that such passive variables and broadcast transitions do not exist in the original concurrent programs to be verified, but are introduced after predicate abstraction of those programs as presented in [11]. They are essential to capture the behaviour of the processes existing in the system other than the process that actually executes a transition (resp. passive and active processes).

Example 7 (Broadcast transition). Consider a concurrent program with shared and local variables $\texttt{s}$ and $\texttt{l}$, the assignment transition $\texttt{t}_0 :: \texttt{pc}_1 \rightarrow \texttt{pc}_2 : \texttt{s} := \texttt{l}$ and the mixed predicate $\texttt{mx} :: (\texttt{s} = \texttt{l})$ to be used for the predicate abstraction. Recall from Sec. 5.1 that such predicates are called *mixed predicates* as they contain both local and shared variables. Each process will have its own copy of a mixed predicate, similar to local predicates. However, unlike local predicates, mixed predicates are updated only by broadcast transitions. Before the assignment $\texttt{t}_0$, consider two passive processes P_1 and P_2 having mixed predicates $\texttt{mx}_{p_1} :: (\texttt{s} = \texttt{l}_{p_1})$ evaluates to $\texttt{tt}$ and $\texttt{mx}_{p_2} :: (\texttt{s} = \texttt{l}_{p_2})$ evaluates to $\texttt{ff}$ and an active process P_a having $\texttt{mx}_{p_a} :: (\texttt{s} = \texttt{l}_a)$ evaluates to $\texttt{tt}$. The active process will execute $\texttt{t}_0$, hence, $\texttt{s} = \texttt{l}_a$ will hold after the transition. At this point, $\texttt{mx}_{p_1}$ will hold only if $\texttt{l}_a = \texttt{l}_{p_1}$. But, $\texttt{mx}_2$ will *not* hold. In fact, after the transition, all passive processes will be notified to update their corresponding mixed predicate w.r.t their own valuation and that of the active process. This corresponds to a broadcast (More details in Ex. 8).

Syntax and semantics of boolean programs. We describe the syntax of boolean programs in Fig. 6. Variables $\texttt{s}^b$ and $\texttt{l}^b$ are some variables respectively in S^b and L^b. Variables $\texttt{v}_1^b, \ldots, \texttt{v}_n^b$ are pairwise different and belong to $S^b \cup L^b \cup L_p^b$. Predicate π is in $\texttt{predsOf}^{S^b \cup L^b}$ (i.e., a boolean combination of boolean variables in $S^b \cup L^b$) and predicates π_i, π_i' are in $\texttt{predsOf}^{S^b \cup L^b \cup L_p^b}$. By construction, the predicate abstraction of concurrent programs [11] does not involve assignments of passive variables to non-passive ones.

We describe semantics of boolean programs in Fig. 7. We add the superscript b to mean the boolean program variant. For instance, we use $\mathbb{S}^b, \mathbb{L}^b$ and $\mathbb{M}^b$ to respectively mean the sets of shared states, local states and processes configurations of boolean programs. The $\texttt{ch}\left(\pi, \pi'\right)$ operator evaluates to $\texttt{tt}$ if π evaluates to $\texttt{tt}$, to $\texttt{ff}$ if π' evaluates to $\texttt{tt}$, and non-deterministically to either $\texttt{tt}$ or $\texttt{ff}$ otherwise. Apart from the fact that all variables are now boolean and that we make use of the $\texttt{ch}$ operator, the main difference of Fig. 7 with Fig. 4 is the $\texttt{assign}$ statement as it may involve passive variables in order to capture broadcasts.

Example 8 ($\texttt{ch}$ operator). Consider once again Ex. 7. Using $\texttt{mx}$, the assignment transition $\texttt{t}_0$ will be abstracted to $\texttt{t}_0^b :: \texttt{pc}_1 \rightarrow \texttt{pc}_2 : \texttt{mx}, \texttt{mx}_p := \texttt{tt}, \texttt{ch}\left(\texttt{mx}_p \wedge \texttt{mx}, \texttt{mx}_p \oplus \texttt{mx}\right)$ ($\oplus$ is exclusive-or). Based on the semantics of $\texttt{ch}$ operator, the variable $\texttt{mx}_p$ will evaluate to $\texttt{tt}$ if both $\texttt{mx}_p$ and $\texttt{mx}$ held before the assignment, will evaluate to $\texttt{ff}$ if $\texttt{mx}_p \oplus \texttt{mx}$ held before the assignment, and evaluates to a non deterministic boolean value otherwise.

Let $\texttt{v}_1^b, \ldots \texttt{v}_n^b := \texttt{ch}\left(\pi_1, \pi_1'\right), \ldots \texttt{ch}\left(\pi_n, \pi_n'\right)$ be an assignment. When describing the semantics in Fig. 7, we write $\left(\mathbb{s}^b, \mathbb{l}^b, \mathbb{l}_p^b\right) \xrightarrow[\texttt{abst}_\Pi(P)]{\texttt{v}_1^b, \ldots \texttt{v}_n^b := \texttt{ch}\left(\pi_1, \pi_1'\right), \ldots \texttt{ch}\left(\pi_n, \pi_n'\right)} \left(\mathbb{s}^{b\prime}, \mathbb{l}^{b\prime}, \mathbb{l}_p^{b\prime}\right)$ in order to mean that a process other than the one performing the assignment (i.e. a passive process) with local state $\mathbb{l}_p^b$ can move to $\mathbb{l}_p^{b\prime}$ when the active process moves from $\mathbb{l}^b$ to $\mathbb{l}^{b\prime}$ and the shared state changes from $\mathbb{s}^b$ to $\mathbb{s}^{b\prime}$. The new shared state and the new local state of the active process are obtained from their older versions according to (3) and (4). The local state $\mathbb{l}_p^{b\prime}$ of the passive process is obtained as follows. First, we change the domain of $\mathbb{l}_p^b$ from L^b to L_p^b and obtain $\mathbb{l}_{p,1}^b$ as described in (5). Then we apply the assignment to obtain $\mathbb{l}_{p,2}^b$ accroding to (6). Finally, we obtain $\mathbb{l}_p^{b\prime}$ by changing the domain of $\mathbb{l}_{p,2}^b$ from $\mathbb{L}_p^b$ back to $\mathbb{L}^b$. Notice that at least one $\mathbb{l}_p^{b\prime}$ exists for each $\mathbb{l}_p^b$. Intuitively, this step corresponds to updating, during a broadcast, where an active process changes local state together with possibly many passive processes.

$$\mathbb{s}^{b\prime} = \mathbb{s}^b[\{\texttt{v}_i^b \leftarrow \texttt{ch}\left(\pi_i, \pi_i'\right)[\mathbb{s}^b, \mathbb{l}^b] | \texttt{v}_i^b \in S^b \wedge i : 1 \leq i \leq n\}] \quad (3)$$

$$\mathbb{l}^{b\prime} = \mathbb{l}^b[\{\texttt{v}_i^b \leftarrow \texttt{ch}\left(\pi_i, \pi_i'\right)[\mathbb{s}^b, \mathbb{l}^b] | \texttt{v}_i^b \in L^b \wedge i : 1 \leq i \leq n\}] \quad (4)$$

$$\mathbb{l}_{p,1}^b = \{\texttt{l}_p^b \leftarrow \mathbb{l}_p^b(\texttt{l}^b) | \texttt{l}^b \in L^b\} \quad (5)$$

$$\mathbb{l}_{p,2}^b = \mathbb{l}_{p,1}^b[\{\texttt{v}_i^b \leftarrow \texttt{ch}\left(\pi_i, \pi_i'\right)[\mathbb{s}^b, \mathbb{l}^b, \mathbb{l}_{p,1}^b] | \texttt{v}_i^b \in L_p^b \wedge i : 1 \leq i \leq n\}] \quad (6)$$

Example 9 (Updating passive local state). Consider once again Ex. 7, Ex. 8, the active local state $\mathbb{l}^b = \{\texttt{mx} \leftarrow \texttt{ff}\}$ and a passive local state $\mathbb{l}_p^b = \{\texttt{mx} \leftarrow \texttt{tt}\}$. The assignment in boolean transition $\texttt{t}_0^b$ involves both variables $\texttt{mx}$ and $\texttt{mx}_p$. We use $\mathbb{l}^b$ and $\mathbb{l}_p^b$ for evaluating respectively $\texttt{mx}$ and $\texttt{mx}_p$, but the domain of the passive local state $\mathbb{l}_p^b$ is L^b based on the syntax of boolean concurrent programs. We need to change its domain to L_p^b in order to be able

to distinguish between passive and non-passive variables. By doing this, we obtain $\mathbb{l}^b_{p,1} = \{\mathtt{mx}_p \leftarrow \mathtt{tt}\}$. Then, we update $\mathbb{l}^b_{p,1}$ by the result of the assignment and will have $\mathbb{l}^b_{p,2} = \{\mathtt{mx}_p \leftarrow \mathtt{ff}\}$. Finally, we obtain $\mathbb{l}^{b\prime}_p$ by changing the domain of $\mathbb{l}^b_{p,2}$ from $\mathbb{L}^b_p$ back to $\mathbb{L}^b$ as $\mathbb{l}^{b\prime}_p = \{\mathtt{mx} \leftarrow \mathtt{ff}\}$.

An $\mathtt{abst}_{\Pi}(P)$ run $(\mathbb{s}^b_0, \mathbb{m}^b_0)t^b_1 \cdots t^b_n(\mathbb{s}^b_n, \mathbb{m}^b_n)$ is a configuration starting sequence of alternating transitions and configurations. This run is considered *feasible* if we have that $(\mathbb{s}^b_i, \mathbb{m}^b_i) \xrightarrow[\mathtt{abst}_{\Pi}(P)]{t^b_{i+1}} (\mathbb{s}^b_{i+1}, \mathbb{m}^b_{i+1})$ for each $i : 0 \leq i < n$ and $\mathbb{s}^b_0$, $\mathbb{m}^b_0$ are in $\mathbb{S}^b_{init}$ and $\mathbb{M}^b_{init}$ respectively. Configurations $(\mathbb{s}^b_i, \mathbb{m}^b_i)$, for $i : 0 \leq i \leq n$, are then said to be reachable.

Evaluating (counting) predicates in $\mathtt{abst}_{\Pi}(P)$. Given a shared configuration $\mathbb{s}^b$, we write $\mathtt{originOf}(\mathbb{s}^b)$ to mean the predicate $\bigwedge_{\mathtt{s}^b \in S^b}(\mathbb{s}^b(\mathtt{s}^b) \Leftrightarrow \mathtt{originOf}(\mathtt{s}^b))$. We write $\mathtt{originOf}(\mathbb{l}^b)$ to mean $\bigwedge_{\mathtt{l}^b \in L^b}(\mathbb{l}^b(\mathtt{l}^b) \Leftrightarrow \mathtt{originOf}(\mathtt{l}^b))$. Observe that $vars(\mathtt{originOf}(\mathbb{s}^b)) \subseteq S$ and the variables $vars(\mathtt{originOf}(\mathbb{l}^b)) \subseteq S \cup L$. We abuse notation and write $\mathbb{s}^b[\mathbb{s}]$ (resp. $\mathbb{l}^b[\mathbb{s}, \mathbb{l}]$) to mean that $\mathtt{originOf}(\mathbb{s}^b)[\mathbb{s}]$ (resp. $\mathtt{originOf}(\mathbb{l}^b)[\mathbb{s}, \mathbb{l}]$) holds.

Let π be a predicate in $\mathtt{predsOf}^{@PC \cup \Pi}$, where all boolean variables are either predicates in Π or of the form $@\mathtt{pc}$ for some $\mathtt{pc} \in PC$. We write $\pi[\mathbb{s}^b, (\mathtt{pc}, \mathbb{l}^b)]$ to mean the boolean value obtained by evaluating the result of replacing each boolean variable $@\mathtt{pc}'$ with $\mathtt{tt}$ if $\mathtt{pc} = \mathtt{pc}'$ and with $\mathtt{ff}$ otherwise, and by replacing each π' in Π with $\mathbb{l}^b(\mathtt{v}^b)$ or $\mathbb{s}^b(\mathtt{v}^b)$ where $\mathtt{originOf}(\mathtt{v}^b) = \pi'$.

We can now evaluate a counting variable $(\pi)^{\#}$, with $\pi \in \mathtt{predsOf}^{@PC \cup \Pi}$, wrt. a configuration $(\mathbb{s}^b, \mathbb{m}^b)$. We do this by counting the number of process states $(\mathtt{pc}, \mathbb{l}^b)$ in $\mathbb{m}^b$ for which $\pi[\mathbb{s}^b, (\mathtt{pc}, \mathbb{l}^b)]$ holds (see Eq. 7). We can also replace each counting variable $(\pi)^{\#}$ in a counting predicate ω with its value in $(\mathbb{s}^b, \mathbb{m}^b)$, and each shared predicate in Π with its value in $\mathbb{s}^b$ (see Eq. 8). Observe that the obtained expression might still involve shared variables as these can participate in comparisons with counting variables. Such comparisons do not correspond to any predicate in Π and can therefore not be mapped.

$$(\pi)^{\#}[\mathbb{s}^b, \mathbb{m}^b] = \sum_{\{(\mathtt{pc}, \mathbb{l}^b) \mid \pi[\mathbb{s}^b, (\mathtt{pc}, \mathbb{l}^b)]\}} \mathbb{m}^b((\mathtt{pc}, \mathbb{l}^b)) \tag{7}$$

$$\omega[\mathbb{s}^b, \mathbb{m}^b] = \omega[\{(\pi)^{\#} \leftarrow (\pi)^{\#}[\mathbb{s}^b, \mathbb{m}^b] | (\pi)^{\#} \in vars(\omega)\}] \tag{8}$$

Relation between P and $\mathtt{abst}_{\Pi}(P)$. We let $\mathbb{m}^b[\mathbb{s}, \mathbb{m}]$ mean that there is a bijection h from $\mathbb{m} = [\sigma_1, \sigma_2, \ldots \sigma_n]$ to $\mathbb{m}^b = [\sigma'_1, \sigma'_2, \ldots \sigma'_n]$ s.t. for each $\mathtt{i}$ in $[1, n]$, if $\sigma_i = (\mathtt{pc}, \mathbb{l})$ then $\sigma'_{h(i)} = (\mathtt{pc}, \mathbb{l}^b)$ and $\mathbb{l}^b[\mathbb{s}, \mathbb{l}]$. The *concretization* of an $\mathtt{abst}_{\Pi}(P)$ configuration $(\mathbb{s}^b, \mathbb{m}^b)$ is $\gamma\left((\mathbb{s}^b, \mathbb{m}^b)\right) = \left\{(\mathbb{s}, \mathbb{m}) | \mathbb{s}^b[\mathbb{s}] \wedge \mathbb{m}^b[\mathbb{s}, \mathbb{m}]\right\}$. The *abstraction* of $(\mathbb{s}, \mathbb{m})$ is the singleton $\alpha((\mathbb{s}, \mathbb{m})) = \left\{(\mathbb{s}^b, \mathbb{m}^b) | \mathbb{s}^b[\mathbb{s}] \wedge \mathbb{m}^b[\mathbb{s}, \mathbb{m}]\right\}$. We initialize variables in $\mathtt{abst}_{\Pi}(P)$ such that for each pair $(\mathbb{s}_{init}, \mathbb{m}_{init})$ of P, there are $(\mathbb{s}^b_{init}, \mathbb{m}^b_{init})$ that are initial in $\mathtt{abst}_{\Pi}(P)$ so that $\alpha((\mathbb{s}_{init}, \mathbb{m}_{init})) = \left\{(\mathbb{s}^b_{init}, \mathbb{m}^b_{init})\right\}$. The abstraction $\alpha(\rho)$ of a P run $\rho = (\mathbb{s}_0, \mathbb{m}_0)t_1 \ldots t_n(\mathbb{s}_n, \mathbb{m}_n)$ is the set of $\mathtt{abst}_{\Pi}(P)$ runs $\left\{(\mathbb{s}^b_0, \mathbb{m}^b_0)t^b_1 \ldots t^b_n(\mathbb{s}^b_n, \mathbb{m}^b_n)\right\}$ where $\alpha((\mathbb{s}_i, \mathbb{m}_i)) = \left\{(\mathbb{s}^b_i, \mathbb{m}^b_i)\right\}$ and $t^b_i = \mathtt{abst}_{\Pi}(t_i)$. Concretizations of abstract runs are defined in a straightforward manner.

Example 10. Consider the program in Fig. 1, its corresponding abstraction wrt. the set of predicates $\Pi = \{\mathtt{read\_leq\_0}, \mathtt{wait\_leq\_count}\}$ in Fig. 5, and the feasible run in Example 4. The initial shared state $\mathbb{s}_0$ defined as $\{\mathtt{count} \leftarrow 1, \mathtt{wait} \leftarrow 0, \mathtt{cross} \leftarrow 0, \mathtt{read} \leftarrow 0\}$ in original program will be encoded as the boolean shared state $\mathbb{s}^b_0 = \{\mathtt{read\_leq\_0} \leftarrow \mathtt{tt}, \mathtt{wait\_leq\_count} \leftarrow \mathtt{tt}\}$ in the boolean program. We have that $\mathtt{originOf}(\mathbb{s}^b_0) = (((\mathtt{read} \leq 0) = \mathtt{tt}) \wedge ((\mathtt{wait} \leq \mathtt{count}) = \mathtt{tt})))$. Since there are no local variables in this example, we get that $\alpha((\mathbb{s}_0, \mathbb{m}_0)) = \left\{(\mathbb{s}^b_0, \mathbb{m}_0)\right\}$. Many other states have the same encoding, e.g. $\mathbb{s}_1 = \{\mathtt{count} \leftarrow 10, \mathtt{wait} \leftarrow 2, \mathtt{cross} \leftarrow 0, \mathtt{read} \leftarrow 0\}$ which satisfies $\mathbb{s}^b_0[\mathbb{s}_1]$, although it is not initial in $\mathtt{prog}$. We get that $\gamma\left((\mathbb{s}^b_0, \mathbb{m}_0)\right) = \{(\mathbb{s}_0, \mathbb{m}_0), (\mathbb{s}_1, \mathbb{m}_0), \ldots\}$.

Definition 1 (predicate abstraction). Let the abstraction of the program $P = (S, L, T)$ wrt. Π be the boolean program $\mathtt{abst}_{\Pi}(P) = \left(S^b, L^b, T^b\right)$. The abstraction is said to be effective and sound if $\mathtt{abst}_{\Pi}(P)$ can be effectively computed and the abstract run in the singleton $\alpha(\rho)$ of any feasible P run ρ is $\mathtt{abst}_{\Pi}(P)$ feasible.

5.2 Encoding into a counter machine

Assume a program $P = (S, L, T)$, a set of predicates $\Pi_0 \subseteq \mathtt{predsOf}_{\mathtt{exprsOf}(S \cup L)}$ and two counting predicates, an invariant ω_{inv} in the set $\mathtt{predsOf}_{\mathtt{exprsOf}(S \cup \Omega_{PC,S,L})}$ and a target predicate ω_{trgt} in $\mathtt{predsOf}_{\mathtt{exprsOf}(\Omega_{PC,S,L})}$. In the following, we write $\mathtt{abst}_{\Pi}(P) = \left(S^b, L^b, T^b\right)$ to mean the abstraction of P with respect to the predicates Π defined as

$$\Pi = \cup_{(\pi)^{\#} \in vars(\omega_{inv}) \cup vars(\omega_{trgt})} \mathtt{comparisonsOf}(\pi) \cup \Pi_0$$

Intuitively, this step results in the formulation of a state reachability problem of a counter machine $enc(\mathtt{abst}_{\Pi}(P))$ that captures reachability of abstractions of ω_{trgt} configurations with $\mathtt{abst}_{\Pi}(P)$ runs that take into account the invariant ω_{inv}.

A tuple $M = (Q, C, \Delta, Q_{init}, \mathbb{C}_{init}, q_{trgt})$ is a counter machine where Q is a finite set of states, C is a finite set of counters (i.e., variables ranging over the natural numbers $\mathbb{N}$), Δ is a finite set of transitions, $Q_{init} \subseteq Q$ is a set of initial states, $\mathbb{C}_{init}$ 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

$$\frac{(\mathbb{s}^b,\mathbb{l}^b,\mathbb{m}^b) \overset{\texttt{stmt}}{\underset{\texttt{abst}_{\Pi}(P)}{\succ\!\!\longrightarrow}} (\mathbb{s}^{b'},\mathbb{l}^{b'},\mathbb{m}^{b'})}{(\mathbb{s}^b,(\texttt{pc},\mathbb{l}^b)\oplus\mathbb{m}^b) \overset{(\texttt{pc}\to\texttt{pc}':\texttt{stmt})}{\underset{\texttt{abst}_{\Pi}(P)}{\longrightarrow}} (\mathbb{s}^{b'},(\texttt{pc}',\mathbb{l}^{b'})\oplus\mathbb{m}^{b'})} : \textit{transition} \qquad \frac{\pi[\mathbb{s}^b,\mathbb{l}^b] \text{ is } \texttt{true}}{(\mathbb{s}^b,\mathbb{l}^b,\mathbb{m}^b) \overset{\pi}{\underset{\texttt{abst}_{\Pi}(P)}{\succ\!\!\longrightarrow}} (\mathbb{s}^b,\mathbb{l}^b,\mathbb{m}^b)} : \textit{assume}$$

$$\frac{(\mathbb{s}^b,\mathbb{l}^b,\mathbb{m}^b) \overset{\texttt{stmt}}{\underset{\texttt{abst}_{\Pi}(P)}{\succ\!\!\longrightarrow}} (\mathbb{s}^{b'},\mathbb{l}^{b'},\mathbb{m}^{b'}) \text{ and } (\mathbb{s}^{b'},\mathbb{l}^{b'},\mathbb{m}^{b'}) \overset{\texttt{stmt}'}{\underset{\texttt{abst}_{\Pi}(P)}{\succ\!\!\longrightarrow}} (\mathbb{s}^{b''},\mathbb{l}^{b''},\mathbb{m}^{b''})}{(\mathbb{s}^b,\mathbb{l}^b,\mathbb{m}^b) \overset{\texttt{stmt};\texttt{stmt}'}{\underset{\texttt{abst}_{\Pi}(P)}{\succ\!\!\longrightarrow}} (\mathbb{s}^{b''},\mathbb{l}^{b''},\mathbb{m}^{b''})} : \textit{sequence}$$

$$\frac{\mathbb{m}^{b'} = \left((\texttt{pc}_0,\mathbb{l}^b_{init})\oplus\mathbb{m}^b\right) \text{ with } \mathbb{l}^b_{init}\in\mathbb{L}^b_{init}}{(\mathbb{s}^b,\mathbb{l}^b,\mathbb{m}^b) \overset{\texttt{spawn}}{\underset{\texttt{abst}_{\Pi}(P)}{\succ\!\!\longrightarrow}} (\mathbb{s}^b,\mathbb{l}^b,\mathbb{m}^{b'})} : \textit{spawn} \qquad \frac{\mathbb{m}^b = \left((\texttt{pc}_x,\mathbb{l}^{b'})\oplus\mathbb{m}^{b'}\right)}{(\mathbb{s}^b,\mathbb{l}^b,\mathbb{m}^b) \overset{\texttt{join}}{\underset{\texttt{abst}_{\Pi}(P)}{\succ\!\!\longrightarrow}} (\mathbb{s}^b,\mathbb{l}^b,\mathbb{m}^{b'})} : \textit{join}$$

$$\frac{\begin{array}{c}\text{a bijection } h \text{ from } \mathbb{m}^b=[\sigma_1,\sigma_2,\dots\sigma_n] \text{ to } \mathbb{m}^{b'}=[\sigma'_1,\sigma'_2,\dots\sigma'_n] \text{ s.t.}\\ \forall i\in[1,n]\,,\ \text{if } \sigma_i=(\texttt{pc}_p,\mathbb{l}^b_p) \text{ and } \sigma'_{h(i)}=(\texttt{pc}'_p,\mathbb{l}^{b'}_p) \text{ then } \texttt{pc}_p=\texttt{pc}'_p\\ \text{and } (\mathbb{s}^b,\mathbb{l}^b,\mathbb{l}^b_p) \overset{\texttt{v}^b_1,\dots\texttt{v}^b_n:=\texttt{ch}(\pi_1,\pi'_1),\dots\texttt{ch}(\pi_n,\pi'_n)}{\underset{\texttt{abst}_{\Pi}(P)}{\longmapsto}} (\mathbb{s}^{b'},\mathbb{l}^{b'},\mathbb{l}^{b'}_p)\end{array}}{(\mathbb{s}^b,\mathbb{l}^b,\mathbb{m}^b) \overset{\texttt{v}^b_1,\dots\texttt{v}^b_n:=\texttt{ch}(\pi_1,\pi'_1),\dots\texttt{ch}(\pi_n,\pi'_n)}{\underset{\texttt{abst}_{\Pi}(P)}{\succ\!\!\longrightarrow}} (\mathbb{s}^{b'},\mathbb{l}^{b'},\mathbb{m}^{b'})} : \textit{assign}$$

Fig. 7. Semantics of boolean concurrent programs. Executions start from some $\mathbb{m}^b_{init}$ in $\mathbb{M}^b_{init}$.

$$\frac{\delta=(q:op:q') \text{ and } \mathbb{c} \overset{op}{\underset{M}{\succ\!\!\longrightarrow}} \mathbb{c}'}{(q,\mathbb{c}) \overset{\delta}{\underset{M}{\longrightarrow}} (q',\mathbb{c}')} : \textit{transition}$$

$$\frac{}{\mathbb{c} \overset{nop}{\underset{M}{\succ\!\!\longrightarrow}} \mathbb{c}} : \textit{nop} \qquad \frac{\mathbb{c} \overset{op}{\underset{M}{\succ\!\!\longrightarrow}} \mathbb{c}' \text{ and } \mathbb{c}' \overset{op'}{\underset{M}{\succ\!\!\longrightarrow}} \mathbb{c}''}{\mathbb{c} \overset{op;op'}{\underset{M}{\succ\!\!\longrightarrow}} \mathbb{c}''} : \textit{sequence}$$

$$\frac{\exists A.grd[\mathbb{c}]\wedge\mathbb{c}'=\mathbb{c}[\{c_i\leftarrow \texttt{e}_i[\mathbb{c}] | i:1\le i\le n\}]}{\mathbb{c} \overset{grd\Rightarrow(c_1\dots c_n:=\texttt{e}_1\dots\texttt{e}_n)}{\underset{M}{\succ\!\!\longrightarrow}} \mathbb{c}'} : \begin{array}{l}\textit{guarded}\\ \textit{command}\end{array}$$

Fig. 8. Semantics of a counter machine

in the guarded command rule in Fig. 8. A guard *grd* is a predicate in $\texttt{predsOf}_{\texttt{exprsOf}(A\cup C)}$ and a command *cmd* is a multiple assignment $c_1,\dots,c_n := \texttt{e}_1,\dots,\texttt{e}_n$ that involves $\texttt{e}_1,\dots\texttt{e}_n$ in $\texttt{exprsOf}(A\cup C)$ and pairwise different $c_1,\dots c_n$ in C.

A *machine configuration* is a pair $(q,\mathbb{c})$ where q is a state in Q and $\mathbb{c}$ is a mapping $C\to\mathbb{N}$. Semantics are given in Fig. 8. A configuration $(q,\mathbb{c})$ is *initial* if $q\in Q_{init}$ and $\mathbb{c}\in\mathbb{C}_{init}$. An M run ρ_M is a sequence $(q_0,\mathbb{c}_0)\delta_1\dots(q_n,\mathbb{c}_n)$. It is *feasible* if $(q_0,\mathbb{c}_0)$ is initial and $(q_i,\mathbb{c}_i) \overset{\delta_{i+1}}{\underset{M}{\longrightarrow}} (q_{i+1},\mathbb{c}_{i+1})$ for $i:0\le i<n$. The machine state reachability problem is to decide whether there is an M feasible run $(q_0,\mathbb{c}_0)\delta_1\dots(q_n,\mathbb{c}_n)$ s.t. $q_n=q_{trgt}$.

Encoding. We describe in the following a counter machine $enc(\texttt{abst}_{\Pi}(P))$ obtained as an encoding of the boolean program $\texttt{abst}_{\Pi}(P)$. Recall $\texttt{abst}_{\Pi}(P)$ results from the predicate abstraction of the concurrent program P wrt. some initial predicates Π_0 as well as all predicates $\texttt{comparisonsOf}(\pi)$ for every counting variable $(\pi)^{\#}$ in $vars(\omega_{inv})$ and $vars(\omega_{trgt})$. The machine $enc(\texttt{abst}_{\Pi}(P))$ is a tuple $(Q,C,\Delta,Q_{init},\mathbb{C}_{init},q_{trgt})$. A state in Q is either the target state q_{trgt} or is associated to a shared configuration $\mathbb{s}^b$ of $\texttt{abst}_{\Pi}(P)$. We write $q_{\mathbb{s}^b}$ to make the association explicit. There is a bijection that associates a process configuration $(\texttt{pc},\mathbb{l}^b)$ to each counter $c_{(\texttt{pc},\mathbb{l}^b)}$ in C. Only the assign rule makes use of auxiliary variables. In case of broadcasts this rule associates an auxiliary variable $a_{(\texttt{pc},\mathbb{l}^b_p,\mathbb{l}^{b'}_p)}$ to each possible move from process configuration $(\texttt{pc},\mathbb{l}^b_p)$ to $(\texttt{pc},\mathbb{l}^{b'}_p)$.

We will write $\sum_{\{(\texttt{pc},\mathbb{l}^b)|\pi[\mathbb{s}^b,(\texttt{pc},\mathbb{l}^b)]\}} c_{(\texttt{pc},\mathbb{l}^b)}$ to mean the sum of all counters $c_{(\texttt{pc},\mathbb{l}^b)}$ in C such that $\pi[\mathbb{s}^b,(\texttt{pc},\mathbb{l}^b)]$ evaluates to $\texttt{tt}$. We can define the mapping $\mathbb{w}_{\mathbb{s}^b,\omega}$ that maps each counting variable appearing in a counting predicate ω to a corresponding counters sum under $\mathbb{s}^b$. More formally, $\mathbb{w}_{\mathbb{s}^b,\omega}((\pi)^{\#}) = \sum_{\{(\texttt{pc},\mathbb{l}^b)|\pi[\mathbb{s}^b,(\texttt{pc},\mathbb{l}^b)]\}} c_{(\texttt{pc},\mathbb{l}^b)}$ for each counting predicate $(\pi)^{\#}$ in $vars(\omega)$. As a result, $\omega_{trgt}[\mathbb{w}_{\mathbb{s}^b,\omega}]$ is the predicate obtained from ω_{trgt} after replacing all counting variables appearing in it by the corresponding counters sums. Observe that if the predicate ω is in $\texttt{predsOf}_{\texttt{exprsOf}(\Omega_{PC,S,L})}$ as it is the case for ω_{trgt}, then $\omega[\mathbb{w}_{\mathbb{s}^b,\omega}]$ does not mention any shared variables in S. The target predicate ω_{trgt} does not mention any shared variables, because it is to be evaluated in a configura-

$$\frac{t = (\mathtt{pc} \to \mathtt{pc}' : \mathtt{stmt}_{\Pi}) \text{ and } \left[(\mathbb{s}^b, \mathbb{l}^b) : op : (\mathbb{s}^{b\prime}, \mathbb{l}^{b\prime})\right]_{\mathtt{stmt}_{\Pi}}}{(q_{\mathbb{s}^b} : c_{(\mathtt{pc},\mathbb{l}^b)} \geq 1; c^{--}_{(\mathtt{pc},\mathbb{l}^b)}; op; c^{++}_{(\mathtt{pc}',\mathbb{l}^{b\prime})} : q_{\mathbb{s}^{b\prime}}) \in \Delta_t} : transition$$

$$\frac{}{(q_{\mathbb{s}^b} : \omega_{trgt}[\mathbb{w}_{\mathbb{s}^b,\omega_{trgt}}] : q_{trgt}) \in \Delta_{trgt}} : target$$

$$\frac{\left[(\mathbb{s}^b, \mathbb{l}^b) : op : (\mathbb{s}^{b\prime}, \mathbb{l}^{b\prime})\right]_{\mathtt{stmt}} \text{ and } \left[(\mathbb{s}^{b\prime}, \mathbb{l}^{b\prime}) : op' : (\mathbb{s}^{b\prime\prime}, \mathbb{l}^{b\prime\prime})\right]_{\mathtt{stmt}'}}{\left[(\mathbb{s}^b, \mathbb{l}^b) : op; op' : (\mathbb{s}^{b\prime\prime}, \mathbb{l}^{b\prime\prime})\right]_{\mathtt{stmt};\mathtt{stmt}'}} : sequence$$

$$\frac{\pi[\mathbb{s}^b, \mathbb{l}^b]}{\left[(\mathbb{s}^b, \mathbb{l}^b) : nop : (\mathbb{s}^b, \mathbb{l}^b)\right]_{\pi}} : assume \qquad \frac{}{\left[(\mathbb{s}^b, \mathbb{l}^b) : c^{++}_{(\mathtt{pc}_0,\mathbb{l}^b{}_{init})} : (\mathbb{s}^b, \mathbb{l}^b)\right]_{\mathtt{spawn}}} : spawn$$

$$\frac{}{\left[(\mathbb{s}^b, \mathbb{l}^b) : c_{(\mathtt{pc}_x,\mathbb{l}^{b\prime})} \geq 1; c^{--}_{(\mathtt{pc}_x,\mathbb{l}^{b\prime})} : (\mathbb{s}^b, \mathbb{l}^b)\right]_{\mathtt{join}}} : join$$

$$\frac{\begin{array}{c}\mathbb{s}^{b\prime} = \mathbb{s}^b[\{\mathtt{v}^b_i \leftarrow \pi_i[\mathbb{s}^b, (\_, \mathbb{l}^b)] | i : 1 \leq i \leq n\}] \quad \mathbb{l}^{b\prime} = \mathbb{l}^b[\{\mathtt{v}^b_i \leftarrow \pi_i[\mathbb{s}^b, (\_, \mathbb{l}^b)] | i : 1 \leq i \leq n\}] \\ \mathtt{tf} = \left\{ (\mathbb{l}^b_p, \mathbb{l}^{b\prime}_p) | (\mathbb{s}^b, \mathbb{l}^b, \mathbb{l}^b_p) \xrightarrow[\mathtt{abst}_{\Pi}(P)]{\mathtt{v}^b_1,\ldots\mathtt{v}^b_n := \mathtt{ch}(\pi_1,\pi'_1),\ldots\mathtt{ch}(\pi_n,\pi'_n)} (\mathbb{s}^{b\prime}, \mathbb{l}^{b\prime}, \mathbb{l}^{b\prime}_p) \right\}\end{array}}{\left[(\mathbb{s}^b, \mathbb{l}^b) : \left(\begin{array}{l}\bigwedge_{\mathtt{pc}\in PC, \mathbb{l}^b_p \in \mathbb{L}^b} \left(c_{(\mathtt{pc},\mathbb{l}^b_p)} = \sum_{\left\{(\mathtt{pc},\mathbb{l}^{b\prime}_p) | (\mathbb{l}^b_p, \mathbb{l}^{b\prime}_p) \in \mathtt{tf}\right\}} a_{(\mathtt{pc},\mathbb{l}^b_p,\mathbb{l}^{b\prime}_p)}\right) \Rightarrow \\ \mathtt{cmdOf}\left(\left\{c_{(\mathtt{pc},\mathbb{l}^{b\prime}_p)} \leftarrow \sum_{\left\{(\mathtt{pc},\mathbb{l}^b_p) | (\mathbb{l}^b_p, \mathbb{l}^{b\prime}_p) \in \mathtt{tf}\right\}} a_{(\mathtt{pc},\mathbb{l}^b_p,\mathbb{l}^{b\prime}_p)} | \mathtt{pc} \in PC, \mathbb{l}^b_p \in \mathbb{L}^b\right\}\right)\end{array}\right) : (\mathbb{s}^{b\prime}, \mathbb{l}^{b\prime}_p)\right]_{assign}} : assign$$

Fig. 9. Encoding of the transitions of a boolean program (S^b, L^b, T^b), given a counting target ω_{trgt}, to the transitions $\Delta = \cup_{t\in T^b}\Delta_t \cup \Delta_{trgt}$ of a counter machine. In rule *assign*, we write $\mathtt{cmdOf}\left(\left\{c_{(\mathtt{pc},\mathbb{l}^{b\prime})} \leftarrow \mathtt{e}_{(\mathtt{pc},\mathbb{l}^{b\prime})} | \mathtt{pc} \in PC, \mathbb{l}^{b\prime} \in \mathbb{L}^b\right\}\right)$ to mean the multiple assignment that simultaneously assigns each $\mathtt{e}_{(\mathtt{pc},\mathbb{l}^b)}$ to $c_{(\mathtt{pc},\mathbb{l}^b)}$. A transition $\mathbb{c}_1 \xrightarrow[\mathtt{abst}_{\Pi}(P)]{grd \Rightarrow cmd} \mathbb{c}_2$ ensures that there is a mapping $\mathbb{a} : A \to \mathbb{N}$ s.t. for any $c_{(\mathtt{pc},\mathbb{l}^b_p)}$ we have $\mathbb{c}_1(c_{(\mathtt{pc},\mathbb{l}^b_p)}) = \sum_{(\mathbb{l}^b_p,\mathbb{l}^{b\prime}_p)\in\mathtt{tf}} \mathbb{a}(a_{(\mathtt{pc},\mathbb{l}^b_p,\mathbb{l}^{b\prime}_p)})$ and for any $c_{(\mathtt{pc},\mathbb{l}^{b\prime}_p)}$ we have $\mathbb{c}_2(c_{(\mathtt{pc},\mathbb{l}^{b\prime}_p)}) = \sum_{(\mathbb{l}^b_p,\mathbb{l}^{b\prime}_p)\in\mathtt{tf}} \mathbb{a}(a_{(\mathtt{pc},\mathbb{l}^b_p,\mathbb{l}^{b\prime}_p)})$.

tion of the counter machine where no concrete value for shared variables is available. The predicate $\omega_{trgt}[\mathbb{w}_{\mathbb{s}^b,\omega}]$ in $\mathtt{predsOf}_{\mathtt{exprsOf}(C)}$ is used in the target rule of the encoding in Fig. 9.

The set of transitions Δ is exactly the set $\cup_{t\in T^b}\Delta_t \cup \Delta_{trgt}$ as described in Fig. 9. We abuse notation and associate to each statement `stmt` appearing in $\mathtt{abst}_{\Pi}(P)$ the set $enc(\mathtt{stmt})$ of tuples $\left[(\mathbb{s}^b, \mathbb{l}^b) : op : (\mathbb{s}^{b\prime}, \mathbb{l}^{b\prime})\right]_{\mathtt{stmt}}$ generated in Fig. 9. Given a processes configuration $\mathbb{m}^b$, we write $\mathbb{c}_{\mathbb{m}^b}$ to mean the mapping associating $\mathbb{m}^b((\mathtt{pc}, \mathbb{l}^b))$ to each counter $c_{(\mathtt{pc},\mathbb{l}^b)}$ in C. We let Q_{init} be the set $\left\{q_{\mathbb{s}^b_0} | \mathbb{s}^b_0 \in \mathbb{S}^b_{init} \text{ of } \mathtt{abst}_{\Pi}(P)\right\}$, and $\mathbb{C}_{init}$ be the mapping $\{\mathbb{c}_{\mathbb{m}^b} | \mathbb{m}^b((\mathtt{pc}_0, \mathbb{l}^b)) = 1$ for a single $\mathbb{l}^b \in \mathbb{L}^b_{init}$ in $\mathtt{abst}_{\Pi}(P)$ and 0 otherwise$\}$. We associate a program configuration $(\mathbb{s}^b, \mathbb{m}^b)$ to each machine configuration $(q_{\mathbb{s}^b}, \mathbb{c}_{\mathbb{m}^b})$. The machine $enc(\mathtt{abst}_{\Pi}(P))$ encodes $\mathtt{abst}_{\Pi}(P)$ as specified in Lem. 3.

We state in Lem. 3 that the reachability problem of the obtained counter machine is equivalent to the reachability in $\mathtt{abst}_{\Pi}(P)$ of boolean configurations satisfying ω_{trgt}. For this, we make use of Lem. 1 and Lem. 2. Intuitively, these relate executions of the boolean abstraction to the ones of the counter machine encoding.

Lemma 1 (translation). *For any* `stmt` *appearing in the program* $\mathtt{abst}_{\Pi}(P)$, $(\mathbb{s}^b, \mathbb{l}^b, \mathbb{m}^b) \xrightarrow[\mathtt{abst}_{\Pi}(P)]{\mathtt{stmt}} (\mathbb{s}^{b\prime}, \mathbb{l}^{b\prime}, \mathbb{m}^{b\prime})$ *iff* $\mathbb{c}_{\mathbb{m}^b} \xrightarrow[enc(\mathtt{abst}_{\Pi}(P))]{op} \mathbb{c}_{\mathbb{m}^{b\prime}}$ *for a* $\left[(\mathbb{s}^b, \mathbb{l}^b) : op : (\mathbb{s}^{b\prime}, \mathbb{l}^{b\prime})\right]_{\mathtt{stmt}}$ *in* $enc(\mathtt{stmt})$.

Proof. By induction on the number of atomic statements in `stmt`. □

Lemma 2 (translation and abstraction). *Any configuration* $(\mathbb{s}^b, \mathbb{m}^b)$ *is reachable in* $\mathtt{abst}_{\Pi}(P)$ *iff* $(q_{\mathbb{s}^b}, \mathbb{c}_{\mathbb{m}^b})$ *is reachable in* $enc(\mathtt{abst}_{\Pi}(P))$.

Proof. We show that $(\mathbb{s}^b, \mathbb{m}^b)$ is reachable via a run of length n in $\mathtt{abst}_{\Pi}(P)$ iff $(q_{\mathbb{s}^b}, \mathbb{c}_{\mathbb{m}^b})$ is reachable via a run of the same length in $enc(\mathtt{abst}_{\Pi}(P))$. We proceed by induction on the number of P transitions appearing in the runs. By construction, $\mathbb{s}^b_0$ and $\mathbb{m}^b_0$ are initial iff $q_{\mathbb{s}^b_0}$ and $\mathbb{c}_{\mathbb{m}^b_0}$ are also initial. Let $(\mathbb{s}^b, \mathbb{m}^b)$ be a reachable $\mathtt{abst}_{\Pi}(P)$ configuration and $(q_{\mathbb{s}^b}, \mathbb{c}_{\mathbb{m}^b})$ be the corresponding $enc(\mathtt{abst}_{\Pi}(P))$ configuration. We will not consider runs in $enc(\mathtt{abst}_{\Pi}(P))$ that involve Δ_{trgt} transitions as these lead to error states and not to configurations of the form $(q_{\mathbb{s}^b}, \mathbb{c}_{\mathbb{m}^b})$. We show $(\mathbb{s}^b, \mathbb{m}^b) \xrightarrow[\mathtt{abst}_{\Pi}(P)]{(\mathtt{pc}\to\mathtt{pc}':\mathtt{stmt})}$

$(\mathbb{s}^{b'}, \mathbb{m}^{b'})$ iff $(q_{\mathbb{s}^b}, \mathbb{c}_{\mathbb{m}^b}) \xrightarrow[enc(\mathtt{abst}_{\Pi}(P))]{(q_{\mathbb{s}^b} : c_{(\mathtt{pc},\mathbb{l}^b)} \geq 1 \ ; \ c^{--}_{(\mathtt{pc},\mathbb{l}^b)}; op; c^{++}_{(\mathtt{pc}',\mathbb{l}^{b\prime})} : q_{\mathbb{s}^{b\prime}})} (q_{\mathbb{s}^{b\prime}}, \mathbb{c}_{\mathbb{m}^{b\prime}})$ for some $\Big[(\mathbb{s}^b, \mathbb{l}^b) : op : (\mathbb{s}^{b'}, \mathbb{l}^{b'})\Big]_{\mathtt{stmt}}$ in $enc(\mathtt{stmt})$. Semantics of boolean programs in Fig. 7 ensure that $(\mathbb{s}^b, \mathbb{m}^b) \xrightarrow[\mathtt{abst}_{\Pi}(P)]{(\mathtt{pc} \to \mathtt{pc}' : \mathtt{stmt})} (\mathbb{s}^{b'}, \mathbb{m}^{b'})$ iff $\mathbb{m}^b = (\mathtt{pc}, \mathbb{l}^b) \oplus \mathbb{m}^b_1$ and $\mathbb{m}^{b'} = (\mathtt{pc}', \mathbb{l}^{b'}) \oplus \mathbb{m}^{b'}_1$ and $(\mathbb{s}^b, \mathbb{l}^b, \mathbb{m}^b_1) \xrightarrow[\mathtt{abst}_{\Pi}(P)]{\mathtt{stmt}} (\mathbb{s}^{b'}, \mathbb{l}^{b'}, \mathbb{m}^{b'}_1)$. Lem. 1 ensures this is equivalent to $\mathbb{c}_{\mathbb{m}^b_1} \xrightarrow[enc(\mathtt{abst}_{\Pi}(P))]{op} \mathbb{c}_{\mathbb{m}^{b'}_1}$ for some $\Big[(\mathbb{s}^b, \mathbb{l}^b) : op : (\mathbb{s}^{b'}, \mathbb{l}^{b'})\Big]_{\mathtt{stmt}}$ in $enc(\mathtt{stmt})$. Observe that $\mathbb{c}_{\mathbb{m}^b_1} \xrightarrow[enc(\mathtt{abst}_{\Pi}(P))]{op} \mathbb{c}_{\mathbb{m}^{b'}_1}$ is equivalent to $\mathbb{c}_{(\mathtt{pc},\mathbb{l}^b) \oplus \mathbb{m}^b_1} \xrightarrow[enc(\mathtt{abst}_{\Pi}(P))]{c_{(\mathtt{pc},\mathbb{l}^b)} \geq 1; c^{--}_{(\mathtt{pc},\mathbb{l}^b)}; op; c^{++}_{(\mathtt{pc}',\mathbb{l}^{b\prime})}} \mathbb{c}_{(\mathtt{pc}',\mathbb{l}^{b\prime}) \oplus \mathbb{m}^{b'}_1}$. □

Lemma 3 (translation reachability). *Target q_{trgt} is $enc(\mathtt{abst}_{\Pi}(P))$ reachable iff a configuration $(\mathbb{s}^b, \mathbb{m}^b)$ is reachable in $\mathtt{abst}_{\Pi}(P)$ such that $\omega_{trgt}[\mathbb{s}^b, \mathbb{m}^b]$ holds.*

Proof. Lem. 2 ensures that $(\mathbb{s}^b, \mathbb{m}^b)$ is $\mathtt{abst}_{\Pi}(P)$ reachable iff $(q_{\mathbb{s}^b}, \mathbb{c}_{\mathbb{m}^b})$ is $enc(\mathtt{abst}_{\Pi}(P))$ reachable. We conclude by observing that $\omega_{trgt}[\mathbb{s}^b, \mathbb{m}^b]$ holds iff the evaluation of the target predicate in the counter machine holds, i.e. $\omega_{trgt}[q_{\mathbb{s}^b}, \mathbb{c}_{\mathbb{m}^b}]$ holds. □

5.3 Encoding precision

We argue in the following that the obtained counter machine often results in a monotonic transition system for which the reachability problem is decidable. In fact, predicate abstraction forces monotonicity. For example, in Fig. 1, transitions $\mathtt{t}_4$ and $\mathtt{t}_5$ correspond to a barrier which is non-monotonic. But, the abstracted boolean program in Fig. 5 that corresponds to it, consists of only monotonic transitions. This happens because the relation between the number of processes in different program locations and the program variables is lost. This corresponds to a loss of precision that makes it impossible to establish correctness of programs such as the one depicted in Fig. 1. We explain how we do retrieve some of that precision by strengthening the abstraction.

Consider the boolean program $\mathtt{abst}_{\Pi}(P)$ obtained after predicate abstraction. If a configuration $(\mathbb{s}^{b'}, \mathbb{m}^{b'})$ is obtained from $(\mathbb{s}^b, \mathbb{m}^b)$ using some transition, then the same transition can obtain a larger configuration (i.e., has the same shared state $\mathbb{s}^{b'}$ and more processes at the same process states in $\mathbb{m}^{b'}$) than $(\mathbb{s}^{b'}, \mathbb{m}^{b'})$ from any configuration larger than $(\mathbb{s}^b, \mathbb{m}^b)$. Lem. 4 shows that indeed, all transitions in Fig. 9 (except for rule *target*) are monotonic with respect to the ordering $\sqsubseteq$ defined by $(q, \mathbb{c}) \sqsubseteq (q', \mathbb{c}')$ iff $q = q'$ and $\mathbb{c} \preceq \mathbb{c}'$[1].

[1] $\mathbb{c} \preceq \mathbb{c}'$ iff $\mathbb{c}(c) \leq \mathbb{c}'(c)$ for each $c \in C$.

Lemma 4 (monotonicity). *Transitions $(q : op : q')$ generated by all rules in Fig. 9, except for the target rule, are monotonic wrt. $\sqsubseteq$.*

Proof. Let *op* be some operation appearing in a generated transition $(q : op : q')$ of $enc(\mathtt{abst}_{\Pi}(P))$. We say that an operation *op* is monotonic wrt. $\preceq$ if for each $\mathbb{c}_1, \mathbb{c}_2, \mathbb{c}_3$ s.t. $\mathbb{c}_1 \xrightarrow[enc(\mathtt{abst}_{\Pi}(P))]{op} \mathbb{c}_2$ and $\mathbb{c}_1 \preceq \mathbb{c}_3$ there exists an $\mathbb{c}_4$ s.t. $\mathbb{c}_3 \xrightarrow[enc(\mathtt{abst}_{\Pi}(P))]{op} \mathbb{c}_4$ and $\mathbb{c}_2 \preceq \mathbb{c}_4$. 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 $grd \Rightarrow cmd$. The first four cases are straightforward. We show $grd \Rightarrow cmd$ is monotonic. Suppose we are given $\mathbb{c}_1, \mathbb{c}_2, \mathbb{c}_3$ s.t. $\mathbb{c}_1 \xrightarrow[enc(\mathtt{abst}_{\Pi}(P))]{grd \Rightarrow cmd} \mathbb{c}_2$ and $\mathbb{c}_1 \preceq \mathbb{c}_3$. We exhibit a $\mathbb{c}_4$ s.t. $\mathbb{c}_3 \xrightarrow[enc(\mathtt{abst}_{\Pi}(P))]{grd \Rightarrow cmd} \mathbb{c}_4$ and $\mathbb{c}_2 \preceq \mathbb{c}_4$. The operation $grd \Rightarrow cmd$, resulted from the assign rule in Fig. 9. This was defined wrt. to two pairs $(\mathbb{s}^b, \mathbb{l}^b)$ and $(\mathbb{s}^{b'}, \mathbb{l}^{b'})$. We fix these two pairs. By the semantics of counter machines (Fig. 8) and of the translation of the *assign* statements in Fig. 9, the fact that $\mathbb{c}_1 \xrightarrow[enc(\mathtt{abst}_{\Pi}(P))]{grd \Rightarrow cmd} \mathbb{c}_2$, ensures that there is a mapping $\mathbb{a} : A \to \mathbb{N}$ s.t. for any $c_{(\mathtt{pc},\mathbb{l}^b_p)}$ we have $\mathbb{c}_1(c_{(\mathtt{pc},\mathbb{l}^b_p)}) = \sum_{(\mathbb{l}^b_p, \mathbb{l}^{b\prime}_p) \in \mathtt{tf}} \mathbb{a}(a_{(\mathtt{pc},\mathbb{l}^b_p,\mathbb{l}^{b\prime}_p)})$ and for any $c_{(\mathtt{pc},\mathbb{l}^{b\prime}_p)}$ we have $\mathbb{c}_2(c_{(\mathtt{pc},\mathbb{l}^{b\prime}_p)}) = \sum_{(\mathbb{l}^b_p, \mathbb{l}^{b\prime}_p) \in \mathtt{tf}} \mathbb{a}(a_{(\mathtt{pc},\mathbb{l}^b_p,\mathbb{l}^{b\prime}_p)})$. Since $\mathbb{c}_1 \preceq \mathbb{c}_3$, then for all $c_{(\mathtt{pc},\mathbb{l}^b_p)} \in C$, we have that $\mathbb{c}_3(c_{(\mathtt{pc},\mathbb{l}^b_p)}) = \mathbb{c}_1(c_{(\mathtt{pc},\mathbb{l}^b_p)}) + r_{(\mathtt{pc},\mathbb{l}^b_p)} = \sum_{(\mathbb{l}^b_p, \mathbb{l}^{b\prime}_p) \in \mathtt{tf}} \mathbb{a}(a_{(\mathtt{pc},\mathbb{l}^b_p,\mathbb{l}^{b\prime}_p)}) + r_{(\mathtt{pc},\mathbb{l}^b_p)}$ where $r_{(\mathtt{pc},\mathbb{l}^b_p)} \geq 0$. The idea is to send these "excedents" along the enabled transfers. Fix such a $(\mathtt{pc}, \mathbb{l}^b_p)$. By definition of $(\mathbb{s}^b, \mathbb{l}^b, \mathbb{l}^b_p) \xrightarrow[\mathtt{abst}_{\Pi}(P)]{\mathtt{v}^b_1, \ldots \mathtt{v}^b_n := \mathtt{ch}(\pi_1, \pi'_1), \ldots \mathtt{ch}(\pi_n, \pi'_n)} (\mathbb{s}^{b'}, \mathbb{l}^{b'}, \mathbb{l}^{b\prime}_p)$ in Sect. 5.1 and of $\mathtt{tf}$ in Fig. 9, we know there is at least a $\underline{\mathbb{l}^{b\prime}_p}$ such that $(\mathbb{l}^b_p, \underline{\mathbb{l}^{b\prime}_p}) \in \mathtt{tf}$. We define $\mathbb{c}_4(c_{(\mathtt{pc},\mathbb{l}^b_p)}) := \sum_{(\mathbb{l}^b_p, \mathbb{l}^{b\prime}_p) \in \mathtt{tf}} \mathbb{a}'(a_{(\mathtt{pc},\mathbb{l}^b_p,\mathbb{l}^{b\prime}_p)})$ where we have $\mathbb{a}'(a_{(\mathtt{pc},\mathbb{l}^b_p,\mathbb{l}^{b\prime}_p)}) := \mathbb{a}(a_{(\mathtt{pc},\mathbb{l}^b_p,\mathbb{l}^{b\prime}_p)}) + r_{(\mathtt{pc},\mathbb{l}^b_p)} \delta_{\mathbb{l}^b_p, \underline{\mathbb{l}^{b\prime}_p}}$ with $\delta_{\mathbb{l}^b_1, \mathbb{l}^b_2}$ iff $\mathbb{l}^b_1$ is identical to $\mathbb{l}^b_2$. So, $\mathbb{c}_2(c_{(\mathtt{pc},\mathbb{l}^b_p)}) \preceq \mathbb{c}_4(c_{(\mathtt{pc},\mathbb{l}^b_p)})$. We repeat the process for each counter $c_{(\mathtt{pc},\mathbb{l}^b_p)}$ in C. This results in a $\mathbb{c}_4$ where $\mathbb{c}_2 \preceq \mathbb{c}_4$ and for which the same transition (i.e. assign for the two pairs $(\mathbb{s}^b, \mathbb{l}^b)$ and $(\mathbb{s}^{b'}, \mathbb{l}^{b'})$) is possible using the mapping $\mathbb{a}'$. □

In fact, even rule *target* results in monotonic machine transitions for all counting predicates ω_{trgt} that denote upward closed sets of processes (since the intersection of two upward closeds is also upward closed). This is for instance the case of predicates capturing assertion violations but not of those capturing deadlocks (see Sec. 4). An encoding $enc(\mathtt{abst}_{\Pi}(P))$ is said to be monotonic if all its transitions are monotonic. Checking assertion violations on abstractions obtained as in Sec. 5.1 always results in monotonic encodings.

Lemma 5 (decidability). *State reachability of all monotonic encodings is decidable.*

Proof. The ordering $\sqsubseteq$ is a well quasi ordering on the set of configuration of $enc(\mathtt{abst}_{\Pi}(P))$ [10]. Monotonicity of the transition rules means the obtained counter machines result in well structured transition systems [3, 14]. It is well known that state reachability is decidable for such systems. □

Strengthening. 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 such as the one in Fig. 1) are no more guaranteed to be upward closed. This loss of precision makes verification out of the reach of techniques solely based on monotonic encodings. To regain some of the lost precision, we constrain the runs using counting invariants. This is done by strengthening the counter machine transitions in order to only allow configurations allowed by ω_{inv}.

Consider again the program of Fig. 1 with shared state $\mathbb{s} = \{\mathtt{count} \leftarrow 4, \mathtt{wait} \leftarrow 0, \mathtt{cross} \leftarrow 0, \mathtt{read} \leftarrow 0\}$ and process configuration $\mathbb{m} = \{\mathtt{pc}_3 \leftarrow 1, \mathtt{pc}_1 \leftarrow 3\}$. After $\mathtt{t}_4$, $\mathbb{s}' = \{\mathtt{count} \leftarrow 4, \mathtt{wait} \leftarrow 1, \mathtt{cross} \leftarrow 0, \mathtt{read} \leftarrow 0\}$ and $\mathbb{m}' = \{\mathtt{pc}_4 \leftarrow 1, \mathtt{pc}_1 \leftarrow 3\}$. Note that executing transition $\mathtt{t}_5$ (barrier) at $(\mathbb{s}', \mathbb{m}')$ is impossible because of the barrier condition. Abstraction of the program wrt. the predicates $\pi_1 = \mathtt{read\_leq\_0}$, $\pi_2 = \mathtt{wait\_leq\_count}$, $\pi_3 = \mathtt{count\_leq\_wait}$ and $\pi_4 = \mathtt{cross\_leq\_0}$ yields abstract transitions $\mathtt{t}_4^{\mathtt{b}}$ and $\mathtt{t}_5^{\mathtt{b}}$. It turns out that starting from the abstract configuration $(\mathbb{s}^b, \mathbb{m}^b)$ the two transitions can be executed as follows:

$\mathbb{s}^b$				$\mathbb{m}^b$
π_1	π_2	π_3	π_4	pc
true	true	false	true	$\mathtt{pc}_2, \mathtt{pc}_3$
$\Downarrow \mathtt{t}_4^{\mathtt{b}}.\mathtt{wait\_leq\_count} :=$ $\mathtt{ch}(\mathtt{wait\_leq\_count} \wedge \neg\mathtt{count\_leq\_wait}, \mathtt{count\_leq\_wait})$ $\mathtt{count\_leq\_wait} := \mathtt{ch}(\mathtt{count\_leq\_wait}, \mathtt{ff})$				
true	true	true	true	$\mathtt{pc}_2, \mathtt{pc}_4$
$\Downarrow \mathtt{t}_5^{\mathtt{b}}.\mathtt{wait\_leq\_count} \wedge \mathtt{count\_leq\_wait}; \mathtt{cross\_leq\_0} := \mathtt{ff}$				
true	true	true	false	$\mathtt{pc}_2, \mathtt{pc}_5$

The steps above are not feasible in the original program, because $\mathtt{t}_4^{\mathtt{b}}$ changed predicate $\mathtt{count\_leq\_wait}$ to $\mathtt{tt}$ although $\mathtt{count} \leq \mathtt{wait}$ does not hold in the original program. This results in a flase positive.

Our solution to recover some of the lost precision is to strengthen the transitions of the counter machine using counting predicates (i.e. predicates that relate shared variables to the number of processes) that are valid in all runs, for example:
$\omega_{inv} : \mathtt{count} = \sum_{\mathtt{i} \geq 1}(@\mathtt{pc}_\mathtt{i})^{\#} \wedge \mathtt{wait} = \sum_{\mathtt{i} \geq 4}(@\mathtt{pc}_\mathtt{i})^{\#}$.

Note that the counting invariant ω_{inv} as opposed to the target predicates ω_{trgt} (Sec. 5.2) contains shared variables. This is to regain the information about the shared variables that was lost due to abstraction. Given an abstraction of a shared state $\mathbb{s}^b$, we first conjunct ω_{inv} with $\mathtt{originOf}(\mathbb{s}^b)$ and project away the shared variables. We get a predicate that only contains counting variables. Then, we substitute the counting variables with their corresponding counters to obtain a predicate that only mentions counter machine variables. We finally strengthen all counter machine transition that involve the state $q_{\mathbb{s}^b}$.

$$\frac{(q_{\mathbb{s}^b} : op : q_{\mathbb{s}^{b\prime}})}{\left(q_{\mathbb{s}^b} : str\left(\mathbb{s}^b, \omega_{inv}\right); op; str\left(\mathbb{s}^{b\prime}, \omega_{inv}\right) : q_{\mathbb{s}^{b\prime}}\right)}$$

$$\frac{\left(q_{\mathbb{s}^b} : \omega_{trgt}[\mathbb{w}_{\mathbb{s}^b, \omega_{trgt}}] : q_{trgt}\right)}{\left(q_{\mathbb{s}^b} : str\left(\mathbb{s}^b, \omega_{inv}\right); \omega_{trgt}[\mathbb{w}_{\mathbb{s}^b, \omega_{trgt}}] : q_{trgt}\right)}$$

Fig. 10. Strengthening of a counter machine transition given a counting invariant ω_{inv}. For this, let $str\left(\mathbb{s}^b, \omega_{inv}\right) = \left(\exists S.\left(\mathtt{originOf}(\mathbb{s}^b) \wedge \omega_{inv}\right)\right)[\mathbb{w}_{\mathbb{s}^b, \omega_{inv}}]$.

Example 11 (Strengthening). Consider the boolean shared state $\mathbb{s}^b$ with $\mathtt{originOf}(\mathbb{s}^b) = ((\mathtt{read} \leq 0) \wedge (\mathtt{wait} \leq \mathtt{count}) \wedge (\mathtt{count} \leq \mathtt{wait}) \wedge \neg(\mathtt{cross} \leq 0))$. After conjunction with the invariant and projection of the shared variables, we obtain: $\exists S.(\mathtt{originOf}(\mathbb{s}^b) \wedge \omega_{inv}) = ((@\mathtt{pc}_1)^{\#} + (@\mathtt{pc}_2)^{\#} + (@\mathtt{pc}_3)^{\#}) = 0$. After substitution, we get $str\left(\mathbb{s}^{b\prime}, \omega_{inv}\right) = (c_{(\mathtt{pc}_1, \emptyset)} + c_{(\mathtt{pc}_2, \emptyset)} + c_{(\mathtt{pc}_3, \emptyset)} = 0)$. Each counter $c_{(\mathtt{pc}_i, \emptyset)}$ is the counter for processes at location $\mathtt{pc}_i$ where the processes have no local state. In other words, in the corresponding boolean program, there exists no local or mixed predicates. We add the condition $(c_{(\mathtt{pc}_1, \emptyset)} + c_{(\mathtt{pc}_2, \emptyset)} + c_{(\mathtt{pc}_3, \emptyset)} = 0)$ as a statement at the beginning (resp. at the end) of each transitions outgoing from (resp. incoming to) $q_{\mathbb{s}^b}$.

Strengthening is described in Fig. 10. We need Lem. 6 and Lem. 7 in order to establish soundness of the strengthening step in Lem. 8. Lem. 6 states that including local and mixed predicates in the predicate abstraction step makes it possible to track the number of processes satisfying boolean combinations of those predicates in the boolean program abstraction.

Lemma 6 (abstraction and counting variables). *If* $(\mathbb{s}^b, \mathbb{m}^b) \in \alpha((\mathbb{s}, \mathbb{m}))$ *then for each predicate* π *belonging to* $\mathtt{predsOf}_{\mathtt{exprsOf}(S \cup L)}$ *s.t.* $\mathtt{comparisonsOf}(\pi) \subseteq \Pi$, $\sum_{\{(\mathtt{pc}, \mathbb{l}) | \pi[\mathbb{s}, (\mathtt{pc}, \mathbb{l})]\}} \mathbb{m}((\mathtt{pc}, \mathbb{l})) = \sum_{\{(\mathtt{pc}, \mathbb{l}^b) | \pi[\mathbb{s}^b, (\mathtt{pc}, \mathbb{l}^b)]\}} \mathbb{m}^b((\mathtt{pc}, \mathbb{l}^b))$.

Proof. By definition of $\alpha((\mathbb{s}, \mathbb{m}))$, $\mathbb{m}^b[\mathbb{s}, \mathbb{m}]$ holds. This means that there is a bijection h from $\mathbb{m} = [\sigma_1, \sigma_2, \ldots \sigma_n]$ to $\mathbb{m}^b = [\sigma'_1, \sigma'_2, \ldots \sigma'_n]$ s.t. we can associate to each $\sigma_i = (\mathtt{pc}, \mathbb{l})$ in $\mathbb{m}$ a $\sigma_{h(i)} = (\mathtt{pc}', \mathbb{l}^b)$ in $\mathbb{m}^b$ such that $\mathtt{pc} = \mathtt{pc}'$ and $\mathbb{l}^b[\mathbb{s}, \mathbb{l}]$. Let π by a boolean combination of predicates in $\Pi_{mix} \cup \Pi_{loc}$. By construction we have that $\pi[\mathbb{s}^b, (\mathtt{pc}, \mathbb{l}^b)] \Leftrightarrow \pi[\mathbb{s}, (\mathtt{pc}, \mathbb{l})]$. In fact, $\pi[\mathbb{s}^b, (\mathtt{pc}, \mathbb{l}^b)]$ and $\pi[\mathbb{s}, (\mathtt{pc}, \mathbb{l})]$ coincide on each π s.t. $\mathtt{comparisonsOf}(\pi) \subseteq \Pi$. This implies that for π with $\mathtt{comparisonsOf}(\pi) \subseteq \Pi$,

$\sum_{\{(\mathtt{pc},\mathbb{l})|\pi[\mathbb{s},(\mathtt{pc},\mathbb{l})]\}} \mathbb{m}(\mathtt{pc},\mathbb{l}) = \sum_{\{(\mathtt{pc},\mathbb{l}^b)|\pi[\mathbb{s}^b,(\mathtt{pc},\mathbb{l}^b)]\}} \mathbb{m}^b(\mathtt{pc},\mathbb{l}^b)$. □

Lemma 7 (abstraction and strengthening). *Assume* $(\mathbb{s}^b,\mathbb{m}^b) \in \alpha((\mathbb{s},\mathbb{m}))$. *If* $\omega_{inv}[\mathbb{s},\mathbb{m}]$ *holds, then* $\left(\exists S.\left(\mathit{originOf}(\mathbb{s}^b) \wedge \omega_{inv}\right)\right)[\mathbb{m}^b]$ *also holds.*

Proof. We know that $\omega_{inv}[\mathbb{s},\mathbb{m}]$ holds by assumption and $\mathtt{originOf}(\mathbb{s}^b)[\mathbb{s}]$ holds since $(\mathbb{s}^b,\mathbb{m}^b) \in \alpha((\mathbb{s},\mathbb{m}))$. Then, $\left(\mathtt{originOf}(\mathbb{s}^b) \wedge \omega_{inv}\right)[\mathbb{s},\mathbb{m}]$ also holds, and hence the predicate $\left(\left(\mathtt{originOf}(\mathbb{s}^b) \wedge \omega_{inv}\right)[\mathbb{s}]\right)[\mathbb{m}]$. Since for each $(\pi)^{\#}$ appearing in $vars(\omega_{inv})$, $\mathtt{comparisonsOf}(\pi) \subseteq \Pi$, we can replace by Lem. 6 $\mathbb{m}$ for $\mathbb{m}^b$ and obtain the predicate $\left(\left(\mathtt{originOf}(\mathbb{s}^b) \wedge \omega_{inv}\right)[\mathbb{s}]\right)[\mathbb{m}^b]$. Finally, the existence of $\mathbb{s}$ ensures that $\left(\exists S.\left(\mathtt{originOf}(\mathbb{s}^b) \wedge \omega_{inv}\right)\right)[\mathbb{m}^b]$ holds. □

Lemma 8 (soundness of strengthening). *If a* P *configuration satisfying* $\omega_{trgt} \in \mathit{predsOf}_{\mathit{exprsOf}(\Omega_{PC,S,L})}$ *is reachable in* P*, then* q_{trgt} *is reachable in* $enc(\mathtt{abst}_{\Pi}(P))$ *after any strengthening wrt. some* P *invariant* $\omega_{inv} \in \mathit{predsOf}_{\mathit{exprsOf}(S \cup \Omega_{PC,S,L})}$.

Proof. By contradiction. Assume a ω_{trgt} configuration is reachable in P. In addition, assume q_{trgt} is not reachable in a counter machine M obtained as the strengthening (wrt. a P counting invariant ω_{inv}) of an encoding of an abstraction $\mathtt{abst}_{\Pi}(P)$ of P as explained in Section 5.2. Let a feasible run ρ_P without an $enc(\mathtt{abst}_{\Pi}(P))_{str}$ feasible run $\rho_{enc(\mathtt{abst}_{\Pi}(P))_{str}}$ where $enc(\mathtt{abst}_{\Pi}(P))_{str}$ is a strengthening of machine $enc(\mathtt{abst}_{\Pi}(P))$ with respect to an invariant ω_{inv}.

According to Def. 1, for each run ρ_P, a non-empty set $\alpha(\rho_P)$ of $\mathtt{abst}_{\Pi}(P)$ feasible runs exist. Moreover, based on Lem. 3, for each run $\rho_{\mathtt{abst}_{\Pi}(P)} \in \alpha(\rho_P)$ there exists an $enc(\mathtt{abst}_{\Pi}(P))$ feasible run $\rho_{enc(\mathtt{abst}_{\Pi}(P))}$ (before strengthening). So, if the run $\rho_{enc(\mathtt{abst}_{\Pi}(P))_{str}}$ does not exist, it is because the run $\rho_{enc(\mathtt{abst}_{\Pi}(P))}$ was not possible after the strengthening phase.

Let $\rho_{enc(\mathtt{abst}_{\Pi}(P))} = (q_{\mathbb{s}_0^b},\mathbb{c}_{\mathbb{m}_0^b}),\delta_1,\ldots(q_{\mathbb{s}_n^b},\mathbb{c}_{\mathbb{m}_n^b})$ and $\rho_P = (\mathbb{s}_0,\mathbb{m}_0),t_1,\ldots,(\mathbb{s}_n,\mathbb{m}_n)$ with $(\mathbb{s}_i^b,\mathbb{m}_i^b) \in \alpha((\mathbb{s}_i,\mathbb{m}_i))$ for each $i : 0 \le i \le n$. Because $\rho_{enc(\mathtt{abst}_{\Pi}(P))}$ is removed after strengthening, then there exists a step $(q_{\mathbb{s}^b},\mathbb{c}_{\mathbb{m}^b}) \xrightarrow[enc(\mathtt{abst}_{\Pi}(P))]{(q_{\mathbb{s}^b}:op:q_{\mathbb{s}^{b\prime}})} (q_{\mathbb{s}^{b\prime}},\mathbb{c}'_{\mathbb{m}^b})$ in $\rho_{enc(\mathtt{abst}_{\Pi}(P))}$ such that its corresponding step is impossible. According to strengthening rule in Fig. 10, $str\left(\mathbb{s}^b,\omega_{inv}\right) = (\exists S.\mathtt{originOf}(\mathbb{s}^b) \wedge \omega_{inv})[\mathbb{w}_{\mathbb{s}^b,\omega_{inv}}]$. So, the fact that the mentioned step is not possible after strengthening implies that either $(q_{\mathbb{s}^b},\mathbb{c}_{\mathbb{m}^b})$ does not satisfy $str\left(\mathbb{s}^b,\omega_{inv}\right)$, or that $(q_{\mathbb{s}^{b\prime}},\mathbb{c}_{\mathbb{m}^{b\prime}})$ does not satisfy $str\left(\mathbb{s}^{b\prime},\omega_{inv}\right)$. Both alternatives violate the fact that ω_{inv} is an invariant, that $(\mathbb{s},\mathbb{m})$ and $(\mathbb{s}',\mathbb{m}')$ are both reachable, and Lem. 7. □

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

Lemma 9. *State reachability is in general undecidable after strengthening.*

Proof. Sketch. Let $CM = (Q,q_0,\Delta,q_F)$ be an arbitrary two counters Minsky machine. Q is a finite set of states, q_0 and q_F are respectively the initial and the final states. The two counters x_1,x_2 range over the natural numbers and start both from 0. Transitions in Δ are either an increment $(q : x_i^{++} : q')$, a decrement $(q : x_i \ge 1; x_i^{--} : q')$ or a test for zero $(q : x_i = 0 : q')$ for $i \in \{1,2\}$. We describe a concurrent program (in Fig. 11) together with predicates Π_0, a counting invariant ω_{inv} and a counting predicate ω_{trgt}. The obtained strengthened encoding captures the counter machine CM in the sense that solving the reachability of the target state of the encoding is equivalent to checking reachability of q_F for CM, which is undecidable in general.

Each process of the concurrent program manipulates one local variable $\mathtt{tl}$ and four shared variables $\mathtt{c}_1,\mathtt{c}_2,\mathtt{st},\mathtt{ts}$. The shared variable $\mathtt{ts}$ and the local variable $\mathtt{tl}$ range both over the three distinct constants $\mathtt{k}_{\mathtt{pm}},\mathtt{k}_{\mathtt{p1}},\mathtt{k}_{\mathtt{p2}}$. These are used to identify three types of process instances. Exactly one instance (called main instance henceforth) of the process executes transitions starting with the guard $\mathtt{ts} = \mathtt{k}_{\mathtt{pm}}$. The main instance spawns (and joins with) two kinds of auxiliary instances, those that execute the transition with the guard $\mathtt{ts} = \mathtt{k}_{\mathtt{p1}}$ and those that execute the transition with the guard $\mathtt{ts} = \mathtt{k}_{\mathtt{p2}}$. Henceforth $\mathtt{k}_{\mathtt{p1}}$ and $\mathtt{k}_{\mathtt{p2}}$ instances. The variable $\mathtt{st}$ ranges over $|Q|$ distinct constants $\{k_q | q \in Q\}$ and encodes the state $q \in Q$ of the counter machine CM. Finally, the variable $\mathtt{c}_1$ is used to capture the number of $\mathtt{k}_{\mathtt{p1}}$ instances, and $\mathtt{c}_2$ the number of $\mathtt{k}_{\mathtt{p2}}$ instances.

The main instance consists of transitions of the form $(\mathtt{pc}_0 \to \mathtt{pc}_0 : stmt_{op})$ that correspond to each transition $(q : op : q')$ in Δ. More precisely,

1. if op is x_i^{++}, for example $\mathtt{t}_{inc}$ in Fig.11, then $\mathtt{stmt}_{\mathtt{op}} = (\mathtt{ts} = \mathtt{k}_{\mathtt{pm}}; \mathtt{ts} := \mathtt{proc}_{\mathtt{i}};\ \mathtt{st} = \mathtt{k}_{\mathtt{q}}; \mathtt{st} := \mathtt{k}_{\mathtt{q}'};\ \mathtt{c}_{\mathtt{i}} := \mathtt{c}_{\mathtt{i}} + 1; \mathtt{spawn})$;
2. if op is x_i^{--}, for example in $\mathtt{t}_{dec}$ in Fig. 11, then $(\mathtt{ts} = \mathtt{k}_{\mathtt{pm}}; \mathtt{st} = \mathtt{k}_{\mathtt{q}}; \mathtt{st} := \mathtt{t}_{\mathtt{q}'}; \mathtt{c}_2 \ge 1; \mathtt{c}_2 := \mathtt{c}_2 - 1; \mathtt{join})$;
3. if op is $(x_i = 0)$, for example in $\mathtt{t}_{tst}$ in Fig. 11, then $\mathtt{stmt}_{\mathtt{op}} = (\mathtt{ts} = \mathtt{k}_{\mathtt{pm}}; \mathtt{st} = \mathtt{k}_{\mathtt{q}}; \mathtt{s} := \mathtt{k}_{\mathtt{q}'}; \mathtt{c}_1 = 0)$;

Finally, we let $\Pi_0 = \{c_1 = 0, c_2 = 0\} \cup \{st = k_q | q \in Q\}$, $\omega_{trgt} = (\mathtt{st} = \mathtt{k}_{\mathtt{q}})^{\#} \ge 1$ and:

$$\omega_{inv} = \begin{pmatrix} (\mathtt{ts} = \mathtt{k}_{\mathtt{pm}} \wedge (\mathtt{c}_1 = (@\mathtt{pc}_{\mathtt{x}} \wedge \mathtt{tl} = \mathtt{k}_{\mathtt{p1}})^{\#}) \\ \wedge (\mathtt{c}_2 = (@\mathtt{pc}_{\mathtt{x}} \wedge \mathtt{tl} = \mathtt{k}_{\mathtt{p2}})^{\#})) \\ \vee\ (\mathtt{ts} = \mathtt{k}_{\mathtt{p1}} \wedge (\mathtt{c}_1 = 1 + (@\mathtt{pc}_{\mathtt{x}} \wedge \mathtt{tl} = \mathtt{k}_{\mathtt{p1}})^{\#}) \\ \wedge (\mathtt{c}_2 = (@\mathtt{pc}_{\mathtt{x}} \wedge \mathtt{tl} = \mathtt{k}_{\mathtt{p2}})^{\#})) \\ \vee\ (\mathtt{ts} = \mathtt{k}_{\mathtt{p2}} \wedge (\mathtt{c}_1 = (@\mathtt{pc}_{\mathtt{x}} \wedge \mathtt{tl} = \mathtt{k}_{\mathtt{p1}})^{\#}) \\ \wedge (\mathtt{c}_2 = 1 + (@\mathtt{pc}_{\mathtt{x}} \wedge \mathtt{tl} = \mathtt{k}_{\mathtt{p2}})^{\#})) \end{pmatrix}$$

Predicate abstraction will maintain the predicates $c_i = 0$ but will loose their connection with the number of the corresponding instances. Strengthening reestablishes this connection and introduces tests for zero

```
int c1 := 0
int c2 := 0
int st := k_{q0}
int ts := k_pm

process :
int tl := k_pm

// auxiliary processes :
t_pr1 : pc_0 → pc_x :  ts = k_p1; ts := k_pm; tl := k_p1
t_pr2 : pc_0 → pc_x :  ts = k_p2; ts := k_pm; tl := k_p2

// main process :
//(q, x1++, q')
t_inc : pc_0 → pc_0 :  ts = k_pm; ts := k_p1; st = k_q;
                       st := k_q'; c1 := c1 + 1; spawn
//(q, x2 >= 1, x2--, q')
t_dec : pc_0 → pc_0 :  ts = k_pm; st = k_q; st := k_q';
                       c2 ≥ 1; c2 := c2 − 1; join
//(q, x1 = 0, q')
t_tst : pc_0 → pc_0 :  ts = k_pm; st = k_q; st := k_q'; c1 = 0
      ...
```

Fig. 11. Encoding a two counter machine.

on the counters tracking k_{p1}, k_{p2} instances, exactly at the encodings of the transitions t_{tst}. □

5.4 Constrained monotonic abstraction and preorder refinement

This step addresses the state reachability problem for a counter machine $M = (Q, C, \Delta, Q_{init}, \mathbb{C}_{init}, q_{trgt})$. As stated in Lem. 9, this problem is in general undecidable for strengthened encodings. The idea here [6] 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 $\mathbb{c} \preceq \mathbb{c}'$ defined as $\wedge_{c \in C} \mathbb{c}(c) \leq \mathbb{c}'(c)$. Intuitively, $\mathbb{c} \preceq \mathbb{c}'$ holds if $\mathbb{c}'$ can be obtained by "adding more processes to" $\mathbb{c}$. 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 Sec. 5.5). Otherwise, the upward closure $\mathbf{Up}_{\preceq}((q, \mathbb{c}))$ responsible for the spurious trace is identified and an interpolant I (with $vars(I) \subseteq C$) is used to refine the preorder as follows:
$\preceq_{i+1} := \{(\mathbb{c}, \mathbb{c}') | \mathbb{c} \preceq_i \mathbb{c}' \wedge (\mathbb{c} \models I \Leftrightarrow \mathbb{c}' \models I)\}$.
Although stronger, the new preorder is again a well quasi ordering and the trace is guaranteed to be eliminated in the next round. We refer the reader to [5] for more details.

input : A machine $M = (Q, C, \Delta, Q_{init}, \mathbb{C}_{init}, q_{trgt})$ and a preorder $\preceq$
output: `not_reachable` or a *run* $(q_1, \mathbb{c}_1), \delta_1, (q_2, \mathbb{c}_2), \delta_2, \ldots \delta_n, (q_{trgt}, \mathbb{c})$

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

Algorithm 1: Monotonic abstraction. The procedure **Check**$(M, \preceq, \rho)$ checks feasibility of the run ρ in the counter machine M. In case the trace is spurious, it refines the preorder $\preceq$ to eliminate that trace.

Lemma 10 (CMA [5]). *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.*

5.5 Simulation on the original concurrent program

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

example	P	$enc(\texttt{abst}_\Pi(P))$	outer loop		inner loop		time(s)
			num.	preds.	num.	preds.	
cyclic barrier	5:3:12	13:31:79	4	5	5	1	30
		13:4:35	2	2	2	0	15
		13:8:64	4	3	6	2	35
dynamic barrier	5:2:8	8:8:44	3	3	3	0	20
		8:1:14	2	1	2	0	3
		8:1:8	1	0	1	0	1
flag	5:2:9	8:16:123	4	4	6	2	34
		8:8:67	3	2	3	0	5
		8:4:22	3	2	5	2	18
semaphore	4:2:7	7:32:89	5	5	5	0	68
		7:7:24	3	3	3	0	14
		7:16:66	4	4	6	2	17
maximum	5:2:8	18:16:172	5	5	8	3	489
		18:4:51	3	3	3	0	18
		8:8:48	4	3	6	2	21
parent-child	2:3:10	9:16:48	3	4	5	2	76
		9:1:16	2	1	2	0	2
		9:4:35	3	2	3	0	5
as-many	3:2:6	8:4:34	3	2	6	3	68
		8:1:9	2	1	2	0	2
		8:2:8	2	1	2	0	2
locals	6:3:13	14:8:47	3	3	3	0	16
		14:2:24	2	1	2	0	8
		14:16:95	4	4	4	0	58
shareds	7:3:11	13:32:130	4	5	10	6	160
		13:2:21	2	1	2	0	6
		12:2:20	2	1	2	0	17

Table 1: Checking assertion violation and deadlock with PACMAN

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 (predicate abstraction refinement) terminates given the inner loop (monotonic abstraction refinement) terminates. Every iteration of the inner loop terminates.*

Proof. By Lem. 2 and Lem. 6, if the concretization of a run of an encoding is P feasible, then ω_{trgt} is P reachable. Soundness is given by soundness of the predicate abstraction and strengthening (Lem. 8), and soundness of the CMA step (Lem. 10). Termination of each iteration of the inner loop is by well quasi ordering (Lem. 10). Termination of each iteration of the outer loop, given the inner one terminates, is by effectiveness of predicate abstraction (Def. 1) and construction of the encoding (Fig. 9) and strengthening (Fig. 10).

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 cannot express).

6 Experimental results

We report on experiments with our prototype PACMAN(for predicated constrained monotonic abstraction). We have conducted our experiments on an Intel Xeon 2.67GHz processor with 8GB of RAM. The reported examples that require refinements of the natural preorder are challenging for existing techniques; either because the examples require stronger orderings than the usual preorder, or because they involve counting target predicates that are not expressed in terms of violations of program assertions. All predicate abstraction predicates and counting invariants have been derived automatically. For the counting invariants, we implemented a thread modular analysis operating on the polyhedra numerical domain. We make use of several optimizations as explained in the following.

– Invariants and unsatisfiable combinations. We discard boolean mappings corresponding to unsatisfi-

able combinations of predicates, for example $\neg(\texttt{wait} \leq \texttt{count}) \wedge \neg(\texttt{count} \leq \texttt{wait})$ in Fig.1. We also use automatically generated invariants (such as $(\texttt{wait} \leq \texttt{count}) \wedge (\texttt{wait} \geq 0)$ in the same example) to filter the state space. Such heuristics dramatically help our state space exploration algorithm.
– Auxiliary variables. To make the analysis possible, we try to use as few auxiliary variables as possible. An auxiliary variable $a_{(\texttt{pc},\texttt{l}^b,\texttt{l}^{b'})}$ captures how many processes in local state $(\texttt{pc},\texttt{l}^b)$ will move to state $(\texttt{pc},\texttt{l}^{b'})$ because of the broadcast transition due to a multiple assignment. Each time this number is constant, which turns out to often happen, we discard the auxiliary variable $a_{(\texttt{pc},\texttt{l}^b,\texttt{l}^{b'})}$.

For each example, we give the number of transitions and variables both in P and in the resulting counter machine.

The tuples under the P column respectively refer to the size of the original program, i.e. number of variables, procedures and transitions in the original program. The tuples under the $enc\,(\texttt{abst}_{\Pi}(P))$ column refer to the size of the counter machine, i.e. number of counters, states and transitions in the extended counter machine. We also state the number of refinement steps and predicates automatically obtained in both refinement loops. We mention the required total time for the analysis. Tool and examples are accessible online[2].

We report on experiments checking assertion violation and deadlock freedom in Tab.1. For each example, we check three versions, the first one is correct and PACMAN establishes that. The second one is modified so that an assertion can be violated, and the third one is modified so that deadlock is possible. PACMAN for the latter two cases exhibits faulty runs as expected.

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 counting predicates and used them in a framework that combines predicate, counter and constrained 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.

[2] `https://gitlab.ida.liu.se/apv/pacman`

References

1. P. 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*, volume 1633 of *Lecture Notes in Computer Science*, pages 146–159. Springer Berlin Heidelberg, 1999.
2. 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.
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, K. Čerāns, B. Jonsson, and Y.-K. Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160:109–127, 2000.
5. 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.
6. P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 721–736. Springer, 2007.
7. P. A. Abdulla, F. Haziza, and L. Holík. Block me if you can! In *Static Analysis*, pages 1–17. Springer, 2014.
8. K. Bansal, E. Koskinen, T. Wies, and D. Zufferey. Structural counter abstraction. In N. Piterman and S. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *Lecture Notes in Computer Science*, pages 62–77. Springer Berlin Heidelberg, 2013.
9. G. Basler, M. Hague, D. Kroening, C.-H. Ong, T. Wahl, and H. Zhao. Boom: Taking boolean program model checking one step further. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 145–149. Springer Berlin Heidelberg, 2010.
10. L. E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *Amer. J. Math.*, 35:413–422, 1913.
11. A. 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*, volume 6806 of *Lecture Notes in Computer Science*, pages 356–371. Springer Berlin Heidelberg, 2011.
12. J. Esparza, R. Ledesma-Garza, R. Majumdar, P. Meyer, and F. Niksic. An smt-based approach to coverability analysis. In *Computer Aided Verification*, pages 603–619. Springer, 2014.
13. 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.

14. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.
15. Z. Ganjei, A. Rezine, P. Eles, and Z. Peng. Abstracting and counting synchronizing processes. In *Verification, Model Checking, and Abstract Interpretation*, pages 227–244. Springer, 2014.
16. T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Computer Aided Verification*, pages 262–274. Springer, 2003.
17. 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.
18. A. Kaiser, D. Kroening, and T. Wahl. Lost in abstraction: Monotonicity in multi-threaded programs. In P. Baldan and D. Gorla, editors, *CONCUR 2014 Concurrency Theory*, volume 8704 of *Lecture Notes in Computer Science*, pages 141–155. Springer Berlin Heidelberg, 2014.
19. J. Kloos, R. Majumdar, F. Niksic, and R. Piskac. Incremental, inductive coverability. In *Computer Aided Verification*, pages 158–173. Springer, 2013.
20. L. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems (a survey). *Computer Languages, Systems & Structures*, 30(34):139 – 169, 2004. Analysis and Verification.