Almost Upward-Closed is just enough!

Parosh Aziz Abdulla[1], Frédéric Haziza[1], and Lukáš Holík[2]

[1] Uppsala University, Sweden
[2] Brno University of Technology, Czech Republic

Abstract. We present a method for automatic verification of systems with a parameterized number of communicating processes, such as mutual exclusion protocols or agreement protocols. The method handles non well quasi-ordered systems and allows to derive invariants which are not upward-closed. In particular, it allows to model fine-grained global transitions, that is, other processes are not inspected atomically by a given process. We show experimentally the efficiency of the method, on various examples, including a fine-grained model of Szymanski's mutual exclusion protocol which, to the best of our knowledge, cannot be done automatically by other existing methods.

1 Introduction

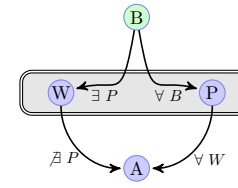
We consider the verification of safety properties for *parameterized systems* with global conditions. They typically consist of an arbitrary number of components (processes) organized according to a certain topology. Processes in the system can perform a transition provided that they satisfy the global condition associated with this transition. For example, in a linear topology, a process (at position i) can perform a transition only if all processes to its left (i.e. with index $j < i$) satisfy a property φ. The latter is an universally quantified transition. For an existentially quantified transition, it is required that *some* (rather than *all*) processes satisfy φ.

The task is to perform *parameterized verification*, i.e., to verify correctness regardless of the number of processes. This amounts to the verification of an infinite family; namely one for each possible size of the system. The term parameterized refers to the fact that the size of the system is (implicitly) a parameter of the verification problem. Parameterized systems arise naturally in the modeling of mutual exclusion algorithms, bus protocols, distributed algorithms, telecommunication protocols, and cache coherence protocols. For instance, the specification of a mutual exclusion protocol may be parameterized by the number of processes that participate in a given session of the protocol. In such a case, it is interesting to verify correctness regardless of the number of participants in a particular session. As usual, the verification of safety properties can be reduced to the problem of checking the reachability of a set of bad configurations (the set of configurations that violate the safety property).

The verification problem is known to be undecidable in general, but decidable for the subclass of parameterized systems which are well quasi-ordered (WQO) [2, 1]. There are numerous methods targeting primarily WQO systems and proven to be complete for them. Universally quantified transitions are not monotonic, and systems with such transitions are not WQO. Some of these methods are however still sound for such systems, and were successfully used to verify many of them, mostly because these systems have an inductive and upward-closed invariant which is strong enough to imply safety. The WQO specialized methods are bound to fail, if they cannot derive such upward-closed invariants. Refer to Section **??** for more details on related works.

Our contribution. We observe that there exists in fact a significant subclass of systems with "almost" upward-closed invariants, that is, invariants which may be expressed as a list of *finite* intersections of upward-closed sets. Let us illustrate such a situation with a simple, yet characteristic, example: the implementation of a barrier synchronization. A process arriving at the barrier must block and cannot proceed until all other processes reach this barrier. The first process at the barrier acts as a *pivot*. All other processes must wait as long as there is a pivot. When all processes have arrived at the barrier, the pivot can proceed, which in turn releases all the waiting processes.

A process can be in state P (pivot), W (waiting), B or A (before and after the barrier). All processes are initially in state B and a bad configuration is detected when there is a process in state A, while there is still a process in state B (cf the figure on the right). It is clear that a pivot P can coexist with a process in state B and/or W, or is alone, i.e. the set of configurations $\{\mathrm{P} \wedge \mathrm{B}^* \wedge \mathrm{W}^*\}$.



We observe that a waiting process W strongly counts on the fact that if there is a delayed process B, then there is *also* a process P. Such a waiting process bases its decision to move through the barrier *solely* on the absence of the pivot (which implies the absence of a process in the state B). In other words, a configuration containing two processes in state W and B *must* also contain a pivot P, i.e. it must be in $\uparrow \{\mathrm{W}^+ \wedge \mathrm{B}^+\} \cap \uparrow \{\mathrm{P}\}$. An intersection of upward-closed sets is not generally upward-closed. Deriving only $\uparrow \{\mathrm{W}^* \wedge \mathrm{B}^*\}$ is clearly not strong enough, because, in such a set, a process in state W could cross the barrier, violating the safety property. We can recognize here a pattern, common for many systems with almost upward-closed invariants: Two processes p_1 and p_2 count collectively on the presence of another process p_3 in a certain state, in order to *not* perform a critical step. Any invariant that does not derive the presence of the blocking process p_3 is (potentially) not strong enough.

We propose a method that can *fully automatically* infer such *context-sensitive* invariants. Of course, since we are dealing with an undecidable problem, the method has its limitations. For instance, we are not able to infer (even to express) invariants which involve equalities of numbers of processes in different states (such invariant could be expressed as an *infinite* intersection of upward closed sets).

Our method retains the simplicity and efficiency of the method from [3] for verification of WQO systems, on which it is based. Thanks to its simplicity, we were able to extend it to handle fine-grained parameterized systems, where the global (quantified) conditions are checked non-atomically. The implementations of checking the global conditions are typically implemented as for-loops ranging over process indices. Verification of systems with non-atomically checked global conditions is significantly harder. The models are closer to the real systems and require to distinguish intermediate states of a for-loop performed by a process. Moreover, any number of processes may be performing a for-loop simultaneously.

To the best of our knowledge, there is no other method which combines the ability to infer non upward-closed invariants with the support of fine-grained modeling. In contrast to many landmark works on parameterized systems [8, 13, 7, 5, 6, 11, 4] which either assume that the universally and existentially quantified transitions are performed atomically or that use simplified models that palliate to the non upward-closed invariants, we were able to verify, e.g., the full fine-grained version of Szymanski's mutual exclusion protocol. The latter was an open problem in parameterized verification. We have moreover tested our method on numerous other case studies, some of which cannot be verified by any other methods we are aware of, and we could observe the promising efficiency of the method. Finally, we show that our method gives rise to a verification scheme which is complete for systems with almost upward-closed invariants.

Outline. To simplify the presentation, we introduce the class of systems we consider in a stepwise manner. First, we consider a basic model in Section 2 where we only allow atomically checked global conditions. We describe how to derive the view abstraction that retains enough information to carry out the verification procedure, fully automatically, in Section 3. Then, we introduce in Section **??** how the abstraction is adapted in order to handle the non-atomic setting. We describe how the abstracted system subtly characterizes the original system, with *in-order* non-atomic checks, while retaining its simplicity. We report on our experimental results in Section 4, and describe related work in Section **??**. Finally, we give some conclusions and directions for future research in Section 5.

2 Coverability in Parameterized Systems on Arrays

We introduce a standard notion of parameterized systems operating on a linear topology, where processes may perform local transitions or universally/existentially guarded transitions — standard model used e.g. in [13, 8, 4, 11].

A parameterized system is a pair $\mathcal{P} = (Q, \Delta)$ where Q is a finite set of *local states* of a process and Δ is a set of *transition rules* over Q. A transition rule is either *local* or *global*. A local rule is of the form $s \to s'$, where the process changes its local state from s to s' independently of the local states of the other processes. A global rule is of the form: **if** $\mathbb{Q}\ j \circ i : S$ **then** $s \to s'$, where $\mathbb{Q} \in \{\exists, \forall\}$, $\circ \in \{<, >, \neq\}$ and $S \subseteq Q$. Here, the i^{th} process checks also the

```
0  flag[i] = 1;
1  for(j=0;j<N;j++){ if(flag[j]≥3) goto 1 ; }
2  flag[i] = 3;
3  for(j=0;j<N;j++){
       if (flag[j] = 1) {
4          flag[i] = 2;
5          for(j=0;j<N;j++){ if(flag[j]==4) goto 7 ; }
6          goto 5 ;
       }
   }
7  flag[i] = 4;
8  for(j=0;j<i;j++){ if(flag[j]≥2) goto 8 ; }
9  /* Critical Section */
10 for(j=i+1;j<N;j++){ if(flag[j]==2||flag[j]==3) goto 10 ; }
11 flag[i] = 0; goto 0 ;
```

Fig. 1: Szymanski's protocol implementation (for process i)

local states of the other processes when it makes the move[3]. For instance, the condition $\forall\, j < i : S$ means that "every process j, with a lower index than i, should be in a local state that belongs to the set S"; the condition $\forall\, j{\neq}i : S$ means that "the local state of all processes, except the i^{th} one, should be one from the set S"; etc...

We use $c[i]$ to denote the state of the i^{th} process within the configuration c. For a configuration c, $i \leq |c|$, and $\delta \in \Delta$, we define the immediate successor $\delta(c,i)$ of c under a δ-move of the i^{th} process such that $\delta(c,i) = c'$ iff $c[i] = s$, $c'[i] = s'$, $c[j] = c'[j]$ for all $j : j \neq i$ and either (i) δ is a local rule $s \to s'$, or (ii) δ is a global rule of the form $\textbf{if}\, \mathbb{Q}\ j \circ i : S\ \textbf{then}\ s \to s'$, where one of the following conditions is satisfied:

- $\mathbb{Q} = \forall$ and for all $j : 1 \leq j \leq |c|$ such that $j \circ i$, it holds that $c[j] \in S$.
- $\mathbb{Q} = \exists$ and there exists $j : 1 \leq j \leq |c|$ such that $j \circ i$ and $c[j] \in S$.

An instance of the *reachability problem* is defined by a parameterized system $\mathcal{P} = (Q, \Delta)$, a regular set $I \subseteq Q^+$ of *initial configurations*, and a set $Bad \subseteq Q^+$ of *bad configurations*. Let $\sqsubseteq$ be the usual *subword relation*, i.e., $u \sqsubseteq s_1 \dots s_n$ iff $u = s_{i_1} \dots s_{i_k}, 1 \leq i_1, \dots, i_k \leq n$ and $i_j < i_{j+1}$ for all $j : 1 \leq j < k$. We assume that Bad is the upward closure $\{c \mid \exists b \in B : b \sqsubseteq c\}$ of a given *finite* set $B \subseteq Q^+$ of *minimal bad configurations*. This is a common way of specifying bad configurations which often appears in practice, see e.g. the running example of Szymanski's mutual exclusion protocol below. We say that $c \in \mathcal{C}$ is *reachable* iff there are $c_0, \dots, c_l \in \mathcal{C}$ such that $c_0 \in I$, $c_l = c$, and for all $0 \leq i < l$, there are $\delta \in \Delta$ and $j \leq |c_i|$ such that $c_{i+1} = \delta(c_i, j)$. We say that the system $\mathcal{P}$ is *safe*

[3] For the sake of presentation, we only consider here a version where a process checks atomically the other processes. The non-atomic case will be introduced in Section **??**.

w.r.t. I and Bad if no bad configuration is reachable, i.e. $\mathcal{R} \cap Bad = \emptyset$, where $\mathcal{R}$ is the set of all reachable configurations. [a note about that we can have more general bad specified as an intersection of upward closed sets?]

We define the *post-image* of a set $X \subseteq \mathcal{C}$ as the set $post(X) = \{\delta(c, i) \mid c \in X, i \leq |c|, \delta \in \Delta\}$. For $n \in \mathbb{N}$ and a set of configurations $X \subseteq \mathcal{C}$, we use X_n to denote its subset $\{c \in X \mid |c| \leq n\}$ of configurations of size up to n.

Example: Szymanski's mutual exclusion protocol. We illustrate the notion of a parameterized systems with the example of Szymanski's mutual exclusion protocol [14]. The protocol ensures exclusive access to a shared resource in a system consisting of an unbounded number of processes organized in an array. The transition rules of the parameterized system are given in Fig. 2. A state of the i^{th} process consists of a program location and a value of the local variable $flag[i]$. Since the value of $flag[i]$ is invariant at each location, states correspond to locations.

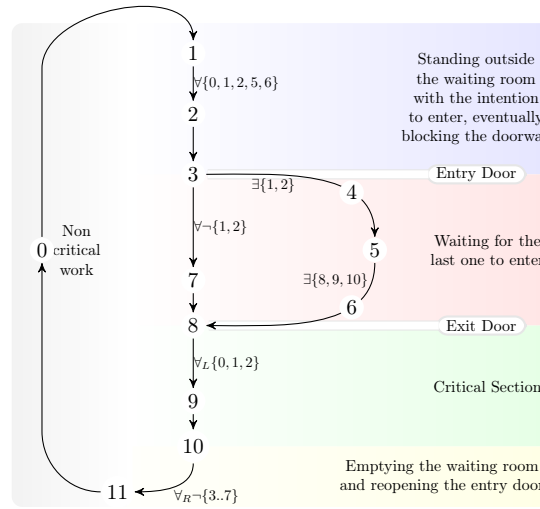


Fig. 2: Szymanski's protocol transition system

A configuration of the induced transition system is a word over the alphabet $\{0, \ldots, 11\}$ of local process states. The task is to check that the protocol guarantees exclusive access to the shared resource regardless of the number of processes. A configuration is considered to be bad if it contains two occurrences of state 9 or 10, i.e., the set of minimal bad configurations B is $\{9\,9, 9\,10, 10\,9, 10\,10\}$. Initially, all processes are in state 0, i.e. $I = 0^+$.

3 Verification method

We present our verification method for the parameterized systems described in Section 2, in the case of *atomically* checked global conditions. Given a parameterized system $\mathcal{P} = (Q, \Delta)$ where Q is a finite set of process states, we first describe the abstraction we use in order to finitely represent configurations of arbitrary size.

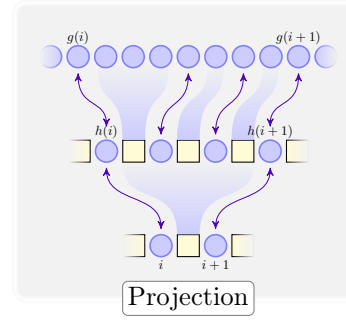
3.1 View Abstraction

A *context-sensitive view* (henceforth only *view*) is a pair $(b_1 \ldots b_k, R_0 \ldots R_k)$, often denoted as $R_0 b_1 R_1 \ldots b_k R_k$, where $b_1 \ldots b_k$ is a configuration and $R_0 \ldots R_k$

is a *context*, such that $R_i \subseteq Q$ for all $i \in [\![0; k]\!]$. We call the configuration $b_1 \dots b_k$ the *base* of the view where k is its *size* and we call each set R_i the i^{th} *context*. We use $\mathcal{V}_k$ to denote the set of views of size up to k.

For $k, n \in \mathbb{N}, k \leq n$, let H_n^k be the set of strictly increasing injections of $[\![0; k+1]\!]$ to $[\![0; n+1]\!]$ such that $1 \leq i < j \leq k \implies 1 \leq h(i) < h(j) \leq n$, $h(0) = 0$ and $h(k+1) = n+1$.

For $h \in H_n^k$ and a configuration $c = q_1 \dots q_n$, we define $\Pi_h(c)$ to be the view $v = R_0 b_1 R_1 \dots b_k R_k$, obtained in the following way: (i) $b_i = q_{h(i)}$ for $i \in [\![1; k]\!]$, (ii) $R_i = \{q_j \mid h(i) < j < h(i+1)\}$ for $i \in [\![0; k]\!]$. Informally, while respecting the order, k elements of c are retained as the base of v, while all other elements are projected away into an (appropriate) context. For a view $v = R_0 b_1 R_1 \dots b_n R_n$, we define the projection $\Pi_h(v)$ as follows. We first define the set $\mathcal{M}(v) = \{(c, g) \mid \Pi_g(c) = v\}$. It represents the set of configurations that can somehow be projected to v. Then, we craftily define $\Pi_h(v) = \Pi_{h \circ g}(c)$ for some $(c, g) \in \mathcal{M}(v)$ (as depicted in the above figure).



The *abstraction function* α_k maps x, a view or a configuration, into the set of *all* its sub-views of size at most k: $\alpha_k(x) = \{\Pi_h(x) \mid h \in H_{|x|}^{\ell}, \ell \leq min(k, |x|)\}$. We lift α_k to sets of views/configurations as usual. Intuitively, we abstract configurations with a *set* of views: the abstraction function extracts *all* subwords from a configuration, while retaining the rest in appropriate contexts.

Entailment. We define the entailement relation on views. Intuitivelly, it compares the restrictions that views impose on the configurations they represent. Let us consider the views $u = R_0 b_1 R_1, \dots, b_k R_k$ and $v = R'_0 b'_1 R'_1, \dots, b'_k R'_k$ of the same size k. We say that u *entails* v, denoted $u \preccurlyeq v$, if $b_i = b'_i$ for all $i \in [\![1; k]\!]$ and $R_i \subseteq R'_i$ for all $i \in [\![0; k]\!]$. For two sets V and W of views, we use $V \preccurlyeq W$ if every $w \in W$ is entailed by some $v \in V$.

Concretization. For every $k \in \mathbb{N}$, the *concretization* function $\gamma_k : 2^{\mathcal{V}_k} \to 2^{\mathcal{C}}$ maps a set of views $V \subseteq \mathcal{V}_k$ into the set of configurations $\gamma_k(V) = \{c \in \mathcal{C} \mid V \preccurlyeq \alpha_k(c)\}$. It is not hard to show that for any k, (α_k, γ_k) forms a Galois connection, as stated by the following lemma, and can be thus used to instantiate an abstract domain.

Lemma 1. *For any k, $V \subseteq \mathcal{V}$ and $C \subseteq \mathcal{C}$, $C \subseteq \gamma_k(V) \iff V \succcurlyeq \alpha_k(C)$.*

Proof. [Should be easy. In fact, we do not need both directions of the implication, but Lukas wants us to have our own Galois connection.]

Precision. For any set $X \in \mathcal{C}$, it is clear that for any k, $\gamma_k(\alpha_k(X)) \supseteq X$. Let us illustrate with the example from Section 1 that the precision of the abstraction increases with k. Consider the set $X = \{\text{PBBW}, \text{PBWW}\}$ of only two configurations. Ignoring the contexts, we get $\alpha_2(X) = \{\text{PB}, \text{BB}, \text{BW}, \text{WW}\}$ and $\alpha_3(X) = \{\text{PBB}, \text{PBW}, \text{BBW}, \text{BWW}, \text{PWW}\}$. We can then observe that $\gamma_2(\alpha_2(X))$

contains WWW, while $\gamma_3(\alpha_3(X))$ does not. In fact, $\gamma_1(\alpha_1(X)) \supseteq \gamma_2(\alpha_2(X)) \supseteq \gamma_3(\alpha_3(X)) \supseteq \ldots \supseteq X$. Furthermore, if we now consider the abstraction *with* contexts (in between brackets), we get $\alpha_2(X) = \{\text{PB[W]}, \text{BB[PW]}, \text{BW[PB]}, \text{BW[PW]}, \text{WW[PB]}\}$. In this case, $\gamma_2(\alpha_2(X))$ does not contain WWW since the view WW[W] is not entailed by any view from X.

Lukas: Let us give an intuitive explanation of how do sets of views encode intersections of upward closed sets of configurations, as claimed in Section 1. ... do this with the same example.

3.2 Procedure

Our verification procedure consists of computing a fixpoint of the standard abstract post-image of the set of initial configurations and checking its intersection with the set of bad configurations. Formally, we compute $\mu X.\, \alpha_k(I) \cup \alpha_k \circ post \circ \gamma_k(X)$ and check that it does not intersect with Bad. Since the precision of the abstraction increases with k and as described in [3], we start with $k = 1$ and iterate the fixpoint computation with larger values of k, until the intersection with Bad is empty.

Symbolic post operator. Since $\gamma_k(X)$ is typically infinite, the abstract post cannot be computed directly and we need a way of computing it symbolically. We show that this can be done by computing the *symbolic* post $\alpha_k \circ spost \circ \gamma_k^{k+1}(X)$ where $\gamma_k^{k+1}(X) = \{v \in \mathcal{V}_{k+1} \mid \alpha_k(v) \succcurlyeq X\}$ is a partial concretization of X to views of size up to $k+1$ and where the post-image *spost* is defined as follows.

For a view $v = (base, ctx)$ and $i \leq |base|$, we first define a symbolic δ-move by the i^{th} process from $base$ as $\delta^{\#}(v, i) = (\delta(base, i), ctx)$ if the δ-transition of that process would not be blocked in case the processes from ctx were present in the base too. It is otherwise undefined. Formally, $\delta^{\#}(v, i) = \Pi_h(\delta(c, h(i)))$ for some $(c, h) \in \mathcal{M}(v)$. Intuitivelly, the context ctx is first "materialized" to create a configuration c, then one of the processes from $base$ performs a transition, and finally the processes that were originally in ctx return back. (Note that the latters are inert and do not perform any transition). Finally, for a set of views X, we define $spost(X) = \{\delta^{\#}(v, i) \mid v \in X, i \leq |v|\}$.

Soundness and precision. It is important to understand why we limit the symbolic post-operator only to views of size $k+1$. Considering the transition system from Section 2, a process needs *at most one* other process as a witness in order to perform its transition (cf existentially-guarded transitions). Therefore, concretizing a view with *several* witnesses will play no role except to be redundant information for the post operation. It is indeed only necessary to "extend" a view with one more process to compute the symbolic successors. The role played by a context is to retain enough information about configurations when they are abstracted into views, such that, when we compute the views in $\gamma_k^{k+1}(X)$, they also retain some information that will *block* some post operation, effectively reducing the risk of running a too coarse over-approximation.

For example, consider again the set $X = \{\text{PBBW}, \text{PBWW}\}$ of only two configurations and the transition system from Figure **??**. We can compute $post(X) = \{\text{PWWW}\}$. Ignoring the contexts, we recall that $V = \alpha_2(X) = \{\text{PB}, \text{BB}, \text{BW}, \text{WW}\}$.

The set $\gamma_2^3(V)$ is rather large but contains surely BBW. The symbolic post-operator leads to the view BBA, since W *can* move to A when there is no P. After abstraction, we notice that the view BA has been computed, but it violates the safety property. However, in case we retain contexts, $V = \alpha_2(X) = \{\text{PB[W]}, \text{BB[PW]}, \text{BW[PB]}, \text{BW[PW]}, \text{WW[PB]}\}$ and $\gamma_2^3(V)$ now contains the view BBW[P]. The symbolic post-operator did not compute the view BA and W is still blocked. We can hence observe that it is more precise in the presence of contexts, because they in fact limit the effect of transitions too loosely enabled.

We show with the following lemma that the symbolic abstract post implements precisely the abstract post: (1)

Lemma 2. *For $k \in \mathbb{N}$ and $X \subseteq \mathcal{V}_k$, $\alpha_k \circ spost \circ \gamma_k^{k+1}(X) \preccurlyeq \alpha_k \circ post \circ \gamma_k(X)$.*(2)

We use $\lfloor V \rfloor$ to denote the set of minimal elements of a set of views V w.r.t. the entailement $\preccurlyeq$. It follows from the definition of γ_k that $\gamma_k(V) = \gamma_k(\lfloor V \rfloor)$ for any set of views, hence sets of configurations can be represented concisely as sets of their $\preccurlyeq$-minimal views. We can therefore use entailment to prune sets of views during the symbolic fixpoint computation. Indeed, instead of $\mu X.\, \alpha_k(I) \cup \alpha_k \circ spost \circ \gamma_k^{k+1}(X)$, we compute the $\preccurlyeq$-equivalent fixpoint $\mu X. \lfloor \alpha_k(I) \cup \alpha_k \circ spost \circ \gamma_k^{k+1}(X) \rfloor$. We describe some accelerations leveraging the power of the entailement in Section 4.

4 Implementation and Experiments

We have implemented a prototype in OCaml based on our method to verify safety properties for a numerous protocols. We report the results running on a 3.1 GHz computer with 4GB memory. Table 1 displays, for various protocols with linear topology (over 2 lines), the running times (in seconds), the final number of views generated ($|V|$) and the number of bases. The first line is the result of the atomic version of the protocol, while the second line represents the non-atomic version. The complete descriptions of the experiments can be found in Appendix A. In most cases, the method terminates almost immediately illustrating the *small model property*: all patterns occur for small instances of the system.

For the first example of Table 1 in the case of non-atomicity, our tool reports the protocol to be *Unsafe*. It is indeed a real error and not an artifact of the over-approximation. The method is sound. In fact, this is also the case when we intentionally tweak the implementation of Szymanski's protocol and force

Table 1: Linear topologies. Atomic vs Non Atomic

Protocol	**Time**	$\lvert bases \rvert$	$\lvert V \rvert$	**status**
Parosh's example	0.005	21	22	*Safe*
	0.006	-	-	*Unsafe*
Burns	0.004	34	34	*Safe*
	0.011	34	64	*Safe*
Dijkstra	0.027	93	93	*Safe*
	0.097	93	222	*Safe*
Szymanski	0.307	100	168	*Safe*
	1.982	100	311	*Safe*
Szymanski (compact)	0.006	48	48	*Safe*
	0.557	48	194	*Safe*
Szymanski (random)	1.156	-	-	*Unsafe*
Bakery	0.001	7	7	*Safe*
	0.006	9	30	*Safe*
Gribomont-Zenner	0.328	119	143	*Safe*

the for-loops to iterate randomly through the indices, in the non-atomic case. The tool reports a trace, that is, a sequence of configurations — here involving only 3 processes — as a witness of an (erroneous) scenario that leads to a violation of the mutual exclusion property.

Table 2: Petri Net with Inhibitor Arcs

Protocol	**Time**	$\|bases\|$	$\|V\|$	**status**
Critical Section with lock	0.001	42	42	*Safe*
Priority Allocator	0.001	33	33	*Safe*
Barrier with Counters	0.001	22	22	*Safe*
Simple Barrier	0.001	7	8	*Safe*
Light Control	0.001	9	15	*Safe*
List with Counter Automata	0.002	20	38	*Safe*

contexts needed

We also ran the method to verify several examples with a multi-set topology: Petri nets with inhibitor arcs. Inhibitor places should retain a context in order to not fire the transition and potentially make the over-approximation too coarse. The bottom part of Table 2 lists examples where the contexts were necessary to verify the protocol, while the top part lists examples that didn't require any context.

Accelerations. We discuss a major acceleration for the procedure in 3.2 which leverages the power of the entailment relation. It is based on the observation that $\mathcal{R}_k$ contains configurations of size k, which can be used as initial input for the procedure. A careful reader will observe that $\alpha_k(\mathcal{R}_k)$ contains views of size (up to) k where all views of size k have empty contexts. These will be useful to cut down on the computations of the symbolic post, since its results will often be views that are already in $\alpha_k(\mathcal{R}_k)$ but with larger contexts, and therefore immediately discardable by entailement. Indeed the successor of a view that is entailed by a view from $\alpha_k(\mathcal{R}_k)$ would not bring any new information to the analysis. It is therefore interesting to seed the fixpoint computation with a larger set than $\alpha_k(I)$, namely $\alpha_k(\mathcal{R}_k)$. Furthermore, it is possible to use $\alpha_{k+1}(\mathcal{R}_{k+1})$ to cut down on the computation of the extensions $\gamma_k^{k+1}(X)$. If a view v from $\gamma_k^{k+1}(X)$ is already entailed by a view w in $\alpha_{k+1}(\mathcal{R}_{k+1})$, it is again possible to safely ignore v, using the same argument above (w has an empty context) and since we already have computed the symbolic post on the view w. Finally, since $\mathcal{R}_k$ and $\mathcal{R}_{k+1}$ are finite, their computation can be done using any procedure for exact state-space exploration and we seed the symbolic fixpoint computation with $\alpha_k(\mathcal{R}_k \cup \mathcal{R}_{k+1})$. The net result of these accelerations is that most experiments happen to be already at fixpoint, which demonstrates the efficiency of the method and that most behaviours are captured by small instances of the system.

Heuristics. If we can empirically observe that $\alpha_2(\mathcal{R}_2 \cup \mathcal{R}_3) = \alpha_2(\mathcal{R}_2 \cup \mathcal{R}_3 \cup \mathcal{R}_4)$, we are probably already at fixpoint. It is therefore interesting to inspect whether

a new view could be inserted while computing the symbolic post on $\gamma_2^3(X)$, even though we ignored contexts. If not, we were indeed at fixpoint already and the invariant we discovered is strong enough to imply safety. If so, we can stop the computations, fetch the views that caused the insertion and remember their contexts for the next round of computations. This heuristic happen to be very successful in the case of Szymanski's protocol (in its non-atomic full version).

On the other hand, this strategy can also be applied in general. We do not remember any contexts, and if the procedure discovers a false-positive, we trace the views that generated it and remember their contexts for the next round of computations, in a CEGAR-like fashion. This will in fact discover necessary contexts little at a time. We are convinced that any smarter counter-example analysis, other than the basic one we implemented, will lead to faster context discovery, as this heuristic is interesting if few views need context. It is however inefficient if it happens that mostly all views need a context (as shown with the ring agreement example). Table 3 presents the results of using the insertion and context discovery heuristics.

Table 3: Leveraging the heuristics

Protocol		**Time**	$\|bases\|$	$\|V\|$	**iteration**
Agreement	with insertion heuristic	8.247	76	199	28
	with all contexts	3.950	76	216	1
	with contexts discovery	166.893	77	121	4
Gribomont-Zenner	with insertion heuristic	0.328	119	143	7
	with all contexts	0.808	119	317	1
	with contexts discovery	50.049	119	217	3
Szymanski, non-atomic	with insertion heuristic	2.053	100	311	26
	with all contexts	48.065	100	771	1
	with contexts discovery	732.643	100	896	7

5 Conclusion and Future Work

We have presented a method for automatic verification of parameterized systems which extends the view abstraction from [3] but alleviates the lack of precision it exhibits on systems with almost upward-closed invariants. This is a unique method that combines the feature of discovering non upward-closed invariant while allowing to model systems with fine-grained transitions.

The method allows to perform parameterized verification by only analyzing a small set of instances of the system (rather than the whole family) and captures the reachability of bad configurations to imply safety. Our algorithm relies on a very simple abstraction function, where a configuration of the system is approximated by breaking it down into smaller pieces. This give rise to a finite

representation of infinite sets of configurations while retaining enough precision. We have proven that the presented algorithm is complete for systems with almost upward-closed invariants. Based on the method, we have implemented a prototype which performs efficiently on a wide range of benchmarks.

We are currently working on extending the method to the case of multi-threaded programs running on machines with different memory models. These systems have notoriously complicated behaviors. Showing that verification can be carried out through the analysis of only a small number of threads would allow for more efficient algorithms for these systems.

References

1. Abdulla, P.A.: Well (and better) quasi-ordered transition systems. Bulletin of Symbolic Logic 16(4), 457–515 (2010)
2. Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: LICS'96. pp. 313–321 (1996)
3. Abdulla, P.A., Haziza, F., s Holík, L.: All for the price of few (parameterized verification through view abstraction). In: 14^{th} International Conference on Verification, Model Checking, and Abstract Interpretation. LNCS, vol. 7737, pp. 476–495 (2013)
4. Abdulla, P.A., Henda, N.B., Delzanno, G., Rezine, A.: Regular model checking without transducers (on efficient verification of parameterized systems). In: TACAS'07. LNCS, vol. 4424, pp. 721–736. Springer (2007)
5. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J.: Regular model checking made simple and efficient. In: Proc. CONCUR 2002, 13^{th} Int. Conf. on Concurrency Theory. LNCS, vol. 2421, pp. 116–130 (2002)
6. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.: Parameterized verification with automatically computed inductive assertions. In: Berry, Comon, Finkel (eds.) Proc. 13^{th} Int. Conf. on Computer Aided Verification. LNCS, vol. 2102, pp. 221–234 (2001)
7. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: CAV'04. LNCS, vol. 3114, pp. 372–386. Springer (2004)
8. Clarke, E., Talupur, M., Veith, H.: Environment abstraction for parameterized verification. In: VMCAI'06. LNCS, vol. 3855, pp. 126–141. Springer (2006)
9. Lamport, L.: A new solution of dijkstra's concurrent programming problem. Commun. ACM 17(8), 453–455 (Aug 1974), `http://doi.acm.org/10.1145/361082.361093`
10. Lynch, N.A., Shamir, B.P.: Distributed algorithms, lecture notes for 6.852, fall 1992. Tech. Rep. MIT/LCS/RSS-20, MIT (1993)
11. Namjoshi, K.S.: Symmetry and completeness in the analysis of parameterized systems. In: VMCAI'07. LNCS, vol. 4349, pp. 299–313. Springer (2007)
12. Nilsson, M.: Regular Model Checking. Ph.D. thesis, Uppsala UniversityUppsala University, Division of Computer Systems, Computer Systems (2005)
13. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. In: Proc. TACAS '01, 7^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. vol. 2031, pp. 82–97 (2001)
14. Szymanski, B.K.: A simple solution to lamport's concurrent programming problem with linear wait. In: Proceedings of the 2nd international conference on Supercomputing. pp. 621–626. ICS '88, ACM, New York, NY, USA (1988), `http://doi.acm.org/10.1145/55364.55425`

A Case studies

A.1 Burns' mutual exclusion

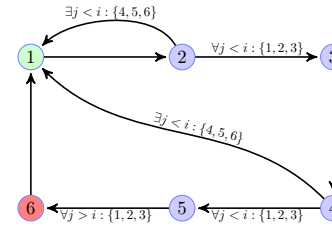
Burns' algorithm [10] implements a mutual exclusion protocol and can be modeled as a parameterized system where the processes are arranged in a linear topology. Each process can communicate with and distinguish its neighbors on its right or its left.

The local state of a process ranges over state $\{1, \ldots, 6\}$ where 6 represents the critical section. Transitions are guarded with conditions on the states of the neighbors, on the right, the left or both and is enabled if the guard is not violated.

Initially, all processes are in state 1. A bad configuration is detected if 2 processes or more are in the critical section, ie if the array contains at least 2 processes in state 6.

The transitions are depicted in the following state diagram. i is the the current process, j is another process and s_j its state.

```
   /* For Process i */
1  flag[i] := 0
2  if ∃j < i : flag[j] = 1 then goto 1
3  flag[i] := 1
4  if ∃j < i : flag[j] = 1 then goto 1
5  await ∀j > i : flag[j] ≠ 1
6  flag[i] := 0; goto 1
```



A.2 Dijsktra's mutual exclusion

Dijsktra's algorithm implements a mutual exclusion protocol and can be modeled as a parameterized system where the processes are arranged in a linear topology. Each process can communicate with its neighbors and check their status.

The algorithm is described in code listing on the right. It makes use of a pointer, i.e. a variable ranging over process indices. We model this pointer by a local boolean variable p for each process state. p is tt iff the pointer points to this current process. When the pointer changes, this information must be passed onto all other processes, which we model as a broadcast transition. Concretely, upon pointer assignment, the current process sets its local variable p to tt and simultaneously sets p to ff in all other processes.

```
   /* For Process i */
1  flag[i] := 1
2  if p ≠ i then
3        await flag[p] = 0 then
4              p = i
5  if ∃j≠i : flag[j] then goto 1
6  flag[i] := 0; goto 1;
```

We denote the state of process as St (resp. Sf) when the process is in state S and the pointer p is tt (resp. ff). The state S of a process ranges over $\{1, \ldots, 6\}$

where 6 represents the critical section. Initially, one process is in state $1t$ and all other processes are in state $1f$. A bad configuration is detected when 2 or more processes are in the critical section, ie when their state is either $6t$ or $6f$.

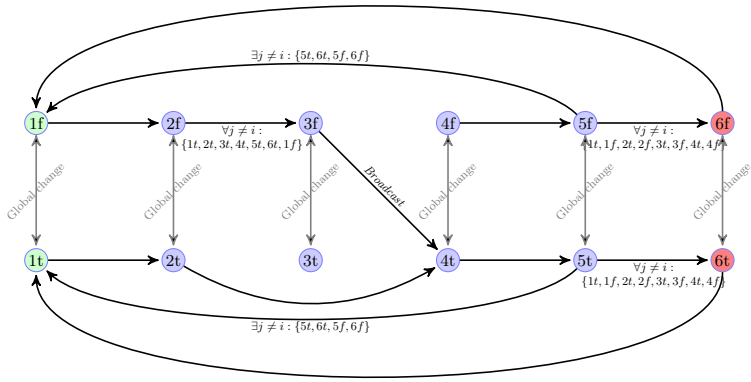


Fig. 3: The transitions of Dijkstra's protocol

A.3 Szymanski's mutual exclusion

The full version of Szymanski's protocol [14] is shown in Section 2. We present here a compact version taken from [6].

The protocol ensures exclusive access to a shared resource in a system consisting of an unbounded number of processes organized in an array. The transition diagram for the ith process is given in Fig. 2. The protocol implements a waiting room with two doors. Processes intending to access the shared resource gather in the waiting room. The last one to enter closes the entry door and opens the exit door. Then, processes access the resource one by one. When all processes have left the waiting room, the exit door closes and the entry door reopens. A process may perform *local* transitions in which it moves independently of the other processes, or transitions that are guarded by *existential* or

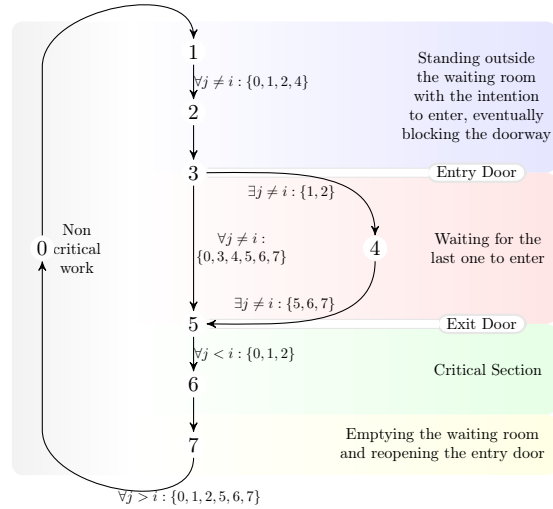
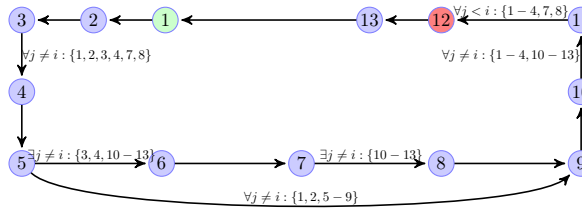


Fig. 4: Szymanski's protocol as a transition system. States correspond to program locations.

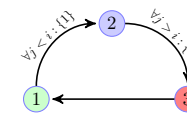
universal conditions on the states of the other processes. In the latter case, the guards may be chosen to refer only to processes on the left or on the right of the current process. Our aim is to prove correctness of the system regardless of the number of processes and check that the protocol guarantees exclusive access to the shared resource. A configuration c is considered to be *bad* if it contains two occurrences of state 6, ie. $66 \sqsubseteq c$. Initially, all processes are in state 0.

A.4 Gribomont-Zenner' mutual exclusion

This algorithm could be seen as a version of Szymanski's algorithm **??**, with transitions that are finer grained in the sense that tests and assignments are split over different atomic transitions. In this model, the local state of a process ranges over state $[1-13]$ where 1 is the initial state and 12 represents the critical section. Configurations not satisfying mutual exclusion are those where at least two processes are at state 12.



A.5 Bakery mutual exclusion



This case study describes a simplified version of the original Bakery algorithm [9]. In this version [12], processes have states that range over $[1-3]$, where 1 is the initial state. A process gets a ticket with a value strictly higher than the ticket value of any process in the queue (transition $1 \to 2$). A process accesses the critical section if it has a ticket with the lowest value among the existing tickets (transition $2 \to 3$). Finally, a process leaves the critical section, freeing its ticket (transition $3 \to 1$). Mutual exclusion violation corresponds to configurations where more than one process is in state 3.

A.6 Parosh's mutual exclusion

This protocol ensures mutual exclusion between processes. Each process has five local states 0, 1, 2, 3, 4 and is initially in state 0. A process in the critical section is at state 4. The set of bad configurations contains exactly configurations with at least two occurrences of state 4.

Processes move from state ⓪ to ①, and then ②. Once the first process is in state ②, it "closes the door" on the processes which are still in ⓪. They can no longer leave state ⓪ until the door is opened again (when no process is in state ② or ③). Moreover, a process is allowed to cross from state ③ to ④ only if there is at least one process still in state ② (i.e., the door is still closed on the processes in state ⓪). This prevents a process to first reach state ④ along with a process to its left starting to move from ⓪ all the way to state ④ (thus violating mutual exclusion). From the set of processes which have left state ⓪ (and which are now in state ① or ②), the leftmost process has the highest priority and it is encoded in the global condition: a process may move from ② to ③ only if all processes on its left are in state ⓪.

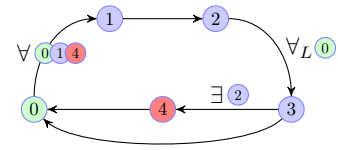
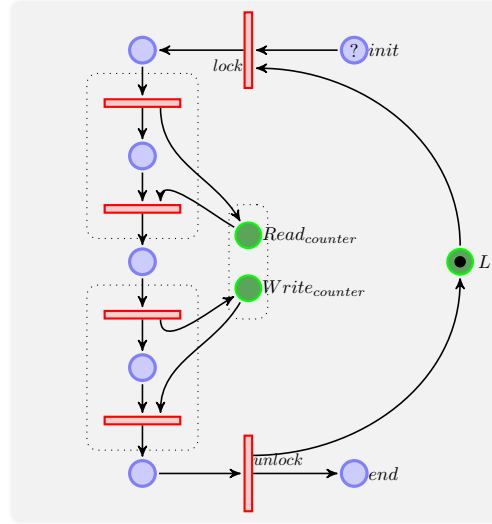


Fig. 5: State diagram (per process).

A.7 Critical section guarded by a lock

We can model the critical section problem by read and write access to a resource shared by multiple processes. There is no particular topology so we will model it as an instance of a parametrized system with multisets. More precisely, as a Petri Net. As the petri net of a concurrent program, described at that level of granularity, can quickly grow in size, we choose a short example: The processes repeatedly grab the lock, increment a *counter* and release the lock. A bad configuration is detected when 2 or more processes are having access to the shared variable simultaneously while one is writing.



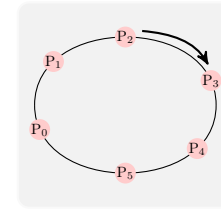
The shared variable *counter* is associated with two places, $Read_{counter}$ and $Write_{counter}$. The tokens of a place in the petri net represent the count of process in a given state, or the available resources. A process places a token in $Read_{counter}$ (resp. $Write_{counter}$) if it is currently accessing the variable *counter* for reading (resp. writing). We model read and write accesses to shared variables with two transitions, denoted by the dotted rectangle in the following figure. There is a place L associated with a lock. Intuitively, if L contains a token, the lock is free, otherwise it is busy. This ensures that only one process can hold

the lock at a time. Note that L is a global variable and that we omit the input places used to balance out the number of tokens in the net.

Initially, the lock is free and the processes are in the initial state *init*. The petri contains then one token in L and the others in *init*. A bad situation is detected when the petri net contains 2 or more tokens in $Write_{counter}$ or when there is one (or more) token in $Read_{counter}$ and one (or more) token in $Write_{counter}$.

A.8 Agreement protocol on a Ring

Interacting peers are organized in a circular pipeline and are given a number. The protocol in place is to ensure that every participant in the ring knows which number in the maximum among the values of the ring members. We model this protocol as an instance of the framework on rings. Each participant in the ring can communicate with its adjacent neighbors. We assume the ring oriented and a member can send messages to its (immediate) "successor" and receive messages from its "predecessor". The reception is usually *blocking* while the sending is not. However, we will model this communication as a rendez-vous.



Initially, all process are in a *dormant* state. One of the process will wake up first. This is the one which initializes the protocol (denoted in the pseudocode as P_0). Every process sends the max between its own value and the value its received from its predecessor (the biggest value it has seen so far). Note that the protocol doesn't not terminate when P_0 receives the biggest value in the ring. It must indeed communicate this value to the others. Each will receive it from its predecessor and only pass it along to its successor (after recording it). The protocol terminates when the predecessor of P_0 receives the biggest value and avoids the re-sending. A bad configuration is detected if one of the participant is in its final state (6 or 17) but has not seen the biggest value go by. We depict the protocol using the following pseudocode. The channel between P_{i-1} and P_i is called `values[n]`. A message in the channel will contain the number the sender sent.

```
   channel values[n](int largest);

   /* For Process P0: initiates the exchanges */
1  int val; // Assume val has been initialized
2  int largest = val; // Initial state
   // Send val to the next process, P1
3  send values[1](largest);
   // Get global largest from Pn-1
4  receive values[0](largest);
   // and pass them on to P1}
5  send values[1](largest); // Pass the final value along
6  // end

   /* For Process P1,...,n-1 */
```

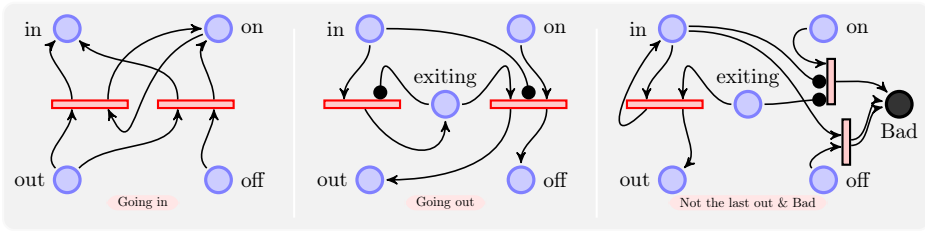
```
10 int val; // Assume val has been initialized
11 int largest; // Initial state
   // Receive the largest value discovered so far
12 receive values[i](largest);
   // then update its val by comparison
13 if (val > largest){ largest = val; }
   // Send the result to the next process
14 send values[(i+1) %n](largest);
   // and then wait to get the global result
15 receive values[i](largest);
16 if (i < n-1) send values[i+1](largest); // Pass it along
17 // end
```

A.9 Light Control

This algorithm implements a simple solution to the light system of an office room. An arbitrary number of people may get in or out of the room. Initially, the light is off and everyone is outside the room. The first person to enter the room turns the light on and the last person to exit the room turns it off.

A bad configuration is detected when the light is on, but there is noone in the room, or when the light is off while there are still people in the room.

We model this algorithm with a Petri Net with Inhibitor arcs. There are 2 places associated with the *on* and *off* status of the light. And there are 2 places associated with the fact that a person is inside or outside the room. An extra place *exiting* is used to check the person who wishes to exit is the last one in the room. Initially, there is a token in the *off* place, and all the other tokens in the outside place. For readibility, we represent the transitions piecewise as follows.



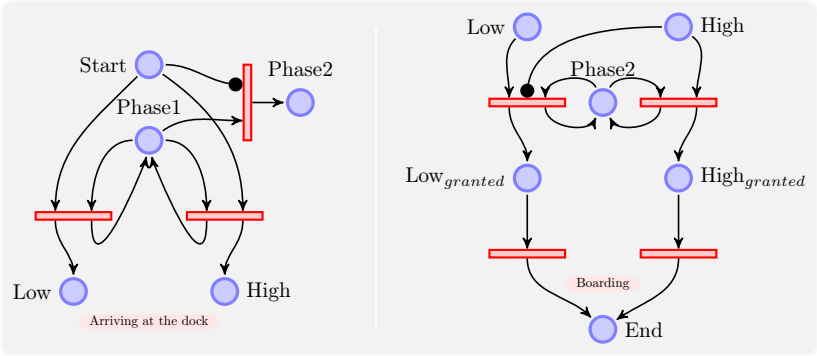
A.10 Priority Allocator

A ferry transports one car at a time from one side of the river to the other side. At the departure, there are 2 queues of cars: One with high priority, one without priority. If there are cars in the high priority lane, the ferry must load them first. If not, it can load a car from the lane with no priority, not even with a first-come-first-served basis.

Initially, the cars must choose their respective lane, but are not allowed to board the ferry, which is not loaded. A bad configuration is detected when the

ferry has loaded a car from the no-priority lane while there are still cars in the high-priority lane.

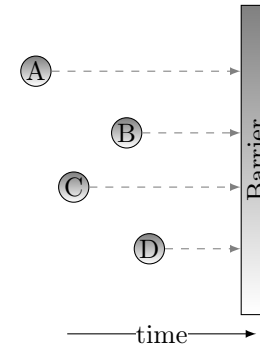
We model this situation with a Petri Net with Inhibitor arcs. There are 2 places associated with the *high* and *low* priority lanes. And there are 2 places associated with the fact that a car has been granted access on the ferry. There is an initialization phase that models how the car get attributed a priority.



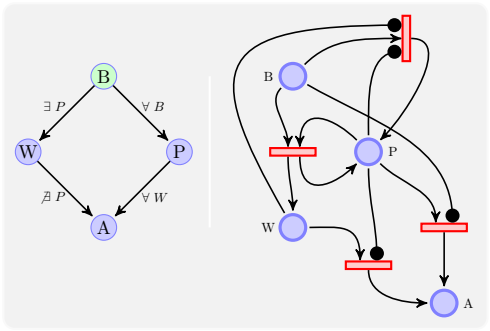
A.11 Simple Barrier

A barrier is a tool to ensure synchronization between threads and allows a programmer to impose restriction for an asynchronous multi-threaded program. A process arriving at the barrier must stop at this point and cannot proceed until all other processes reach this barrier.

The following model represents a barrier with a *pivot*. The first thread at the barrier will take the role of a pivot and all other processes wait as long as there is a pivot. When all processes have arrived at the barrier, the pivot can then proceed, which in turn releases all the waiting processes.



A process can be in state *before* (B), *wait* (W), *pivot* (P) or *after* (A). Initially, all processes are in state B and a bad configuration is detected when there is a process in state A, while there is still a process in state B. The transition diagram and inhibitor net are as follows.



The following invariant can be derived: (i) a pivot P can coexist with a process in state B and/or W, or is alone, i.e. the set of configurations $\{P \wedge B^* \wedge W^*\}$, (ii) a process after the barrier can only coexist with other process after the barrier or still waiting, i.e. the set of configurations $\{W^* \wedge A^+\}$, and (iii) initially, all processes were before the barrier, i.e. $\{B^*\}$.

It is also possible to model it with a linear topology as in Section 1 and 2. In that case, we can consider a non-atomic version, which would not check the state of the other processes all at once.

A typically implementation uses an atomic counter as follows.

```
shared counter   // Initially 0, Ranges over {0,...,n}
shared go        // Atomic bit
local  local.go // A bit

local.go = go; // remembers the current value
< counter = counter + 1; > // atomically increment the
    counter
if ( counter == n ) {    // last to arrive at the barrier
     counter = 0;        // reset
     go = 1 - go;        // notify all
} else {
    while(local.go == go){}; // not the last
}
```

A.12 List with Counter Automata

Consider the following single-threaded program. It scans a list starting from the pointer L and must end when it finds a marker P, placed prior to the start of the program. In this case, the parameter for this parametrized system if the length of the list. The program risks a segmentation fault is the marker P is not found. Our analysis retain the presence of the marker information in a context (and would forget it without context).

```
1  i := L
2  while i ≠ P do
3          i := i.next
4  end
```