Abduction of trap invariants in parameterized systems

Javier Esparza  
Technical University of Munich  
esparza@in.tum.de

Mikhail Raskin  
Technical University of Munich  
raskin@in.tum.de

Christoph Welzel  
Technical University of Munich  
welzel@in.tum.de

In a previous paper we have presented a CEGAR approach for the verification of parameterized systems with an arbitrary number of processes organized in an array or a ring [19]. The technique is based on the iterative computation of parameterized invariants, i.e., infinite families of invariants for the infinitely many instances of the system. Safety properties are proved by checking that every global configuration of the system satisfying all parameterized invariants also satisfies the property; we have shown that this check can be reduced to the satisfiability problem for Monadic Second Order on words, which is decidable.

A strong limitation of the approach is that processes can only have a fixed number of variables with a fixed finite range. In particular, they cannot use variables with range $[0, N-1]$, where $N$ is the number of processes, which appear in many standard distributed algorithms. In this paper, we extend our technique to this case. While conducting the check whether a safety property is inductive assuming a computed set of invariants becomes undecidable, we show how to reduce it to checking satisfiability of a first-order formula. We report on experiments showing that automatic first-order theorem provers can still perform this check for a collection of non-trivial examples. Additionally, we can give small sets of readable invariants for these checks.

1 Introduction

Many distributed systems consist of an arbitrary number of processes executing the same algorithm. For every fixed number of processes the system has a finite state space, and can be verified using conventional model-checking techniques. However, this technique cannot prove that all instances of the system, one for each number of processes, are correct. Parameterized verification designs verification procedures for this task. It has developed a number of techniques, based, among others, on automata theory [4], decidable fragments of first-order logic [7, 31, 32], the theory of well-quasi-orders [1, 5, 20], or the theory of Vector Addition Systems [8, 14, 21, 23].

In recent work, we have investigated the problem of not only proving that a parameterized system satisfies a given safety property, but also providing an explanation of why the property holds in terms of parameterized invariants [9, 10, 19]. A parameterized invariant is an infinite family of invariants, typically one for each number of processes, that can be finitely described in an adequate language.

Assume for simplicity that an instance of the system consists of a tuple of processes $(P_0, \ldots, P_{N-1})$, each of them with states taken from a finite set of states $Q$. We consider variables as processes; for example, a boolean variable is a process with states true and false. A global state of the instance is a tuple $(q_0, \ldots, q_{N-1}) \in Q^N$. Previous research has determined that instances with a small number of processes can often be proved correct using very simple inductive invariants, called trap invariants. A trap invariant is characterized by a tuple $(Q_0, \ldots, Q_{N-1}) \in (2^Q)^N$, called a trap. The invariant states: for every reachable global state $(q_0, \ldots, q_{N-1})$ there exists at least one index $0 \leq i < N$ such that $q_i \in Q_i$. So, loosely speaking, the invariant says that the trap $(Q_0, \ldots, Q_{N-1})$ always remains populated. In the rest of this paper we identify a trap and its associated invariant if it is clear from the context what we refer
Traps in parameterized systems

to. A set of traps proves a safety property if every global state satisfying all the associated trap invariants satisfies the property.

Observe that a trap can be seen as a word of length $N$ over the alphabet $2^Q$. A parametric trap is a regular language over $2^Q$ whose words are traps. The words of a parametric trap can have arbitrary length, and therefore induce invariants of different instances. In [19] we have developed a successful parameterized verification technique, using a CEGAR loop computing a refinement via traps. The loop maintains a set of parametric traps, initially empty. Each iteration considers an instance $\langle P_0, \ldots, P_{N-1} \rangle$ of the system, and consists of three steps:

- **Find.** Find traps $T_1, \ldots, T_k$ whose associated invariants prove the safety property for $\langle P_0, \ldots, P_{N-1} \rangle$.
  This is done using finite-state verification techniques.

- **Abduct.** Abduct each trap $T_j$ (or at least some of them) into a parametric trap $\mathcal{T}_j \subseteq (2^Q)^*$.
  The abduction procedure follows from the very simple structure of traps, and its correctness is guaranteed by a theorem.

- **Check.** Check whether the current collection of parametric traps proves the safety property for every instance of the system. If not, move to the next instance that cannot be proven correct yet, and iterate.

In [9,19] it is shown that this step reduces to the satisfiability of a formula of monadic second-order logic on words, and is, therefore, decidable.

The main restriction of this approach is the fact that the set of states $Q$ of a process cannot depend on the number $N$ of processes. In particular, this forbids the use of variables with range $[0,N-1]$, where $N$ is the number of processes. Using such variables as references to other processes allows to model non-atomic global checks which are essential for many distributed algorithms [26,30]; assume, for example, processes that only have local communication capabilities (as opposed to e.g broadcast communication) but must implement some kind of mutual exclusion. Such variables require that a process has to store the values $\{0,\ldots,N-1\}$ in its state space. In this paper we address particularly the use of iteration variables for non-atomic global checks. Our contribution is as follows:

- We introduce a formal model allowing to describe global checks via looping over all processes, inspecting their local states. This happens non-atomically, i.e., after inspecting, say, process 2, and before inspecting process 3, process 2 may change its local state.

- We generalize the CEGAR loop of [19]. For this, we first show how to generalize the Abduct step to the new formal model. Second, we reduce the Check step to the satisfiability of a formula of a certain fragment of first-order logic.

The price to pay for the added generality is that the fragment is no longer decidable. However, we show that with the help of a theorem prover we can automatically find succinct human-readable correctness proofs for standard mutual exclusion algorithms.

**Related work** Most similar works on parameterized verification focus on systems communicating by rendez-vous and systems with atomic global checks. For example, this is the case for regular model checking [4], parameterized Petri nets [19], component-based systems [9,10] and model checking modulo theories [24]. The latter work admits unbounded domains using decidable theories but, to the best of our knowledge, does not explicitly model non-atomic checks. This, however, is the main focus of the presented work: an explicit extension of the methodology of [19] to account for non-atomic global checks. To this end, we align our work more with the approaches of view-abstraction [2] and symbolic
scheme for over-approximations of parameterized systems [3]. In contrast to these, we see the benefit of our approach in providing readable explanations of safety proofs in form of first-order formulas. The drawback of our approach is a significant longer computation time to establish the desired safety properties. Moreover, the question whether the computed invariants induce the desired safety property is, in general, undecidable (cp. Theorem 2).

2 Formal model

First, we introduce our model of parameterized systems. Let $Q$ and $Vars$ be finite sets of states and variables respectively. Each variable $var \in Vars$ ranges over a finite set $Values_{var}$ of values. A valuation of $Vars$ is a mapping assigning to each $var \in Vars$ an element of $Values_{var}$.

Every $N > 0$ defines an instance of the parameterized system, consisting of $N$ agents with indices $0, 1, \ldots, N - 1$. Every agent has a copy of $Q$ as states, and maintains its own copy of the variables $Vars$. All agents start in the same initial state $q_0$, with the same initial valuation $IVal$.

Agents can execute two types of transitions. Local transitions are of the form

$$\langle origin, \langle var_1 := val_1, \ldots, var_k := val_k \rangle, target \rangle$$

where $origin$ and $target$ are states, $var_1, \ldots, var_k$ are distinct variables, and $val_i \in Values_{var}$, for all $1 \leq i \leq k$. Intuitively, the action allows an agent in state $origin$ to set its own variables $var_1, \ldots, var_k$ to $val_1, \ldots, val_k$ and move to state $target$, all in one single atomic step.

Loop transitions allow agents to loop over all agents inspecting their variables. The inspecting agent or inspector first inspects the agent with index $0$, then the agent with index $1$, and so on, in different atomic steps. Formally, a loop transition has the form

$$\langle origin, \varphi, target_{succ}, target_{fail} \rangle,$$

where $origin$, target$_{succ}$, and target$_{fail}$ are states, and $\varphi$ is a boolean combination of atoms of the form $var = val$ for $var \in Vars$ and $val \in Values_{var}$ and a predicate $self$ which corresponds to the fact that the inspector currently inspects itself. The inspector conducts a sequence of atomic steps. At the $j$-th step the inspector checks if the current valuation of the variables of the agent with index $j$ satisfies $\varphi$. If this is the case and $j = n - 1$ the inspector moves to state $target_{succ}$, if this is the case but $j < n - 1$ the inspector continues to execute the loop transition $t$ and the next time the inspector is scheduled it checks the agent with index $(j + 1)$. Either way, if the agent with index $j$ does not satisfy $\varphi$ the inspector moves to state $target_{fail}$.

Remark 1. All our results can be easily extended to systems with a third kind of transitions of the form $\langle origin, \varphi_1, \varphi_2, target_{succ}, target_{fail} \rangle$, allowing an agent to check if the variables of its left and right neighbors satisfy $\varphi_1$ and $\varphi_2$, respectively, and move to $target_{fail}$ or $target_{succ}$ depending on the result. We omit them for simplicity.

Definition 1 (Parameterized System). A parameterized system over a set $Vars$ of variables is a tuple $\mathcal{S} = \langle Q, q_0, IVal, T_{lo}, T_{lp} \rangle$ where $q_0 \in Q$ is an initial state, $IVal$ is an initial valuation, and $T_{lo}$, $T_{lp}$ are finite sets of local and loop transitions, respectively.

Semantics. A local configuration of an agent is a triple consisting of a location, a valuation, and a pointer. The location is either a state or a loop transition; intuitively, if the current location is a loop
transition \( t \), then the agent is currently an inspector in the “middle” of executing \( t \). The valuation contains the current values of the agent’s variables. The pointer is only important when the current location is a loop transition \( t \); in this case, the pointer is the index of the agent that the inspector is going to inspect next. With a small abuse of language, we say that \( t \) points to this index. A global configuration, or just configuration for short, is a sequence of local configurations, one for each agent.

It is convenient to describe global configurations, and the transitions between them, using a more verbose representation. We first introduce it informally with the help of an example, a very simplified version of Dijkstra’s mutual exclusion algorithm \[12\]. Then we give the formal definition.

**Example 1** (Dijkstra’s algorithm for mutual exclusion). A reduced version of Dijkstra’s algorithm for mutual exclusion can be modeled with states \( \{\text{initial, loop, break, critical, done}\} \) such that initial is the initial state, and one single boolean variable \( b \) with values \( \{\top, \bot\} \) and initial value \( \bot \). There is one loop transition

\[ t_{lp} = (\text{loop, self} \lor b = \bot, \text{critical, break}) \]

and four local transitions:

\[
\begin{align*}
\langle \text{initial, } (b := \top) \rangle, \text{loop} & \quad \langle \text{break, } (b := \bot) \rangle, \text{initial} \\
\langle \text{critical, } () \rangle, \text{done} & \quad \langle \text{done, } (b := \bot) \rangle, \text{initial}.
\end{align*}
\]

Figure [11] describes two steps between configurations of the instance of Dijkstra’s algorithm with three agents. The agents have indices 0, 1, and 2. A local configuration of an agent is represented as a column vector with five components, labeled \( \text{loc, var, } 0.t_{lp}, 1.t_{lp}, 2.t_{lp} \) in the picture. Component \( \text{loc} \) gives the current location of the agent; for example, in the global configuration at the top left of Figure [11], agent 0 is currently in state \( \text{break} \), while agents 1 and 2 are currently executing \( t_{lp} \). Component \( \text{var} \) gives the current value of variable \( b \). The last three components give the next process to be inspected. More precisely, if agent \( i \) is executing \( t_{lp} \), and the next agent it will inspect is the one with index \( j \), then we write an \( \uparrow \)-symbol in the \( i.t_{lp} \) component of vector \( j \). Otherwise, we leave the component blank. In the formal definitions the absence of \( \uparrow \) is denoted with \( \Box \). A (global) configuration is a sequence of three vectors, one for each agent. In the top step of the picture, agent 1 checks variable \( b \) of agent 0. Since the current value of the variable is \( \top \), the agent moves to the failing state, which is \( \text{break} \). In the bottom step of the picture, agent 1 checks variable \( b \) of agent 2. Since the current value is \( \bot \) and \( 2 = N - 1 \), the agent moves to \( \text{critical} \).

Formally, configurations are slotted words.

**Definition 2** (Slotted words). Let \( \text{Sl} = \{s_1, \ldots, s_k\} \) be a finite set of slots, and let \( \Sigma_{s_1}, \ldots, \Sigma_{s_k} \) be finite alphabets, one for each slot. A slotted word over \( \text{Sl} \) is a finite word over the alphabet \( \Sigma_{s_1} \times \cdots \times \Sigma_{s_k} \).

Given a slotted word \( w = a_0 \ldots a_{N-1} \) and \( 0 \leq j < N \), we let \( a_j(s) \in \Sigma_s \) denote the component of slot \( s \) of \( a_j \).

When it is clear from the context, we call a slotted word just a word.

**Definition 3.** Let \( \mathcal{S} = \langle Q, q_0, IVal, T_{loc}, T_{lp} \rangle \) be a parameterized system over \( \text{Vars} = \{\text{var}_1, \ldots, \text{var}_k\} \), where \( T_{lp} = \{t_1, \ldots, t_l\} \), and let \( N > 0 \). The set of slots of \( \mathcal{S} \) is

\[ \text{Sl}_\mathcal{S} = \{\text{loc, var}_1, \ldots, \text{var}_k, 0.t_1, \ldots, 0.t_l, \ldots, (N-1).t_1, (N-1).t_l\} \]

\[ \text{The version still ensures mutual exclusion, but offers no progress guarantees. Our experimental results also include an honest representation of Dijkstra’s algorithm.} \]
with alphabets $\Sigma_{loc} = Q \cup \mathbb{T}_l p$; $\Sigma_{var} = \text{Values}_{var}$, for every $1 \leq i \leq k$; and $\Sigma_{slp} = \{\uparrow, \square\}$ for every $0 \leq i < N$ and $1 \leq j \leq \ell$. With a small abuse of language, we abbreviate the description of $Sl_S$ to

$$Sl_S = \{\text{loc}, \text{Vars}, 0.\mathbb{T}_l p, \ldots, (N-1).\mathbb{T}_l p\}$$

A configuration of the instance of $\mathcal{S}$ with $N$ agents is a slotted word $c_0 \ldots c_{N-1}$ over $Sl_S$ satisfying the following condition for every $0 \leq i, j < N$ and every $t \in \mathbb{T}_l p$: $c_j(i.t) = \uparrow$ iff $c_i(\text{loc}) = t$ and $c_k(i.t) = \square$ for every $k \neq j$. (Intuitively, $c_j(i.t) = \uparrow$ iff agent $i$ is currently executing $t$ and it inspects agent $j$ next.) We call each letter $c_i$ a local configuration.

The initial configuration of the instance is the word $c_0^N$, where $c_0$ is the local configuration given by $c_0(\text{loc}) = q_0$, $c_0(\text{var}_i) = \text{IVal}(\text{var}_i)$ for every $1 \leq i \leq k$, and $c_0(i.t_j) = \square$ for every $0 \leq i < N$ and $1 \leq j \leq \ell$.

The semantics of the instances is given by a transition relation $\vdash$ on configurations on the basis of the transitions of $\mathcal{S}$. Its formal definition is routine, and can be found in Appendix A of the full version [17]. A configuration $c$ is reachable if $c_0 \vdash^* c$, where $C_0$ is the initial configuration of some instance.

### 3 Analysis of instances

For this section, we fix a parameterized system $\mathcal{S}$ and an instance of $\mathcal{S}$ of size $N > 1$ as $\langle C_0, \vdash \rangle$ where $C_0$ denotes the initial configuration, and $\vdash$ its transition relation. Our analysis relies on techniques originally developed for the analysis of specific instances of a system modeled as a Petri net. The technique allows one to compute inductive disjunctive invariants – so called traps – of the set of reachable configurations of the instance [15]. In order to introduce them we first define powerwords.

**Definition 4.** Let $Sl$ be a set of slots with alphabets $\Sigma_s$ for every $s \in Sl$. A slotted powerword is a slotted word over $Sl$ but with alphabets $2^{\Sigma_s}$ for every $s \in Sl$.

A word $w = w_0 \ldots w_{N-1}$ and a powerword $O = O_0 \ldots O_{m-1}$ are compatible if $m = n$. Further, $w$ intersects $O$, denoted $w \cap O$, if $O$ and $w$ are compatible and there is an index $0 \leq i < N$ and a slot $s$ such that $w_i(s) \in O_i(s)$.

Intuitively, a trap of this instance $\langle C_0, \vdash \rangle$ is a powerword of the instance satisfying the following property: the initial configuration intersects the trap, and moreover this property is inductive, i.e., the successor of a configuration intersecting the trap also intersects the trap.

![Figure 1: Steps between configurations of Dijkstra’s algorithm for a 3-agent instance of Example 1.](image-url)

| Index | 0 | 1 | 2 |
|-------|---|---|---|
| loc   | (break) | (break) | (break) |
| var   | $\uparrow$ | $\uparrow$ | $\uparrow$ |
| $0.t_p$ | $\perp$ | $\perp$ | $\perp$ |
| $1.t_p$ | $\perp$ | $\perp$ | $\perp$ |
| $2.t_p$ | $\perp$ | $\perp$ | $\perp$ |

Figure 1: Steps between configurations of Dijkstra’s algorithm for a 3-agent instance of Example 1.
Definition 5 (Trap). Let $\mathcal{S}$ be a parameterized system, and let $\langle C_0, \vdash \rangle$ be an instance of $\mathcal{S}$. A powerword $O$ over $Sl_\mathcal{S}$ is a trap of $\langle C_0, \vdash \rangle$ if $C_0 \cap O$ and all configurations $C, C'$ satisfy: if $C \vdash C'$ and $C \cap O$, then $C' \cap O$.

Example 2. The following powerword is a trap of the 7-agent-instance of Example 1.

| loc | 0 | 1 | 2 | 3 $\{\text{break,loop}\}$ | 4 | 5 $\{\text{break,loop}\}$ | 6 |
|-----|---|---|---|-----------------|---|-----------------|---|
| var | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| 0.tlp | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ |
| 1.tlp | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ |
| 2.tlp | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ |
| 3.tlp | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ |
| 4.tlp | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ |
| 5.tlp | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ | $\{\uparrow\}$ |
| 6.tlp | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |

First observe that the inductiveness of intersecting this powerword cannot be changed by any agent but 3 or 5 executing a transition. Now, if either agent 3 or 5 executes a local transition that sets var to $\perp$ or moves into loop the resulting configuration necessarily marks this trap. If agent 3 (5) moves from state critical to state done then any configuration that intersects this trap before necessarily does not do so at the loc-slot of agent 3 (5) and, hence, the resulting configuration still intersects the trap. It remains to consider the loop transition. If agent 3 or 5 fail the loop inspection then either agent moves into state break which necessarily yields a configuration which intersects the trap. If agent 3 advances its pointer to some agent in $\{0, 1, 2, 3, 4, 5\}$ this yields a configuration intersecting the trap. If agent 3 advances its pointer to agent 6 then agent 5 is inspected in this transition. Since agent 3 only advances if var of agent 5 is currently in state $\perp$ the resulting configuration intersects the trap at the var-slot of agent 5. If agent 3 inspects agent 6 and moves to critical only the state of agent 3 changes from $tlp$ to critical and the 3.tlp-slot of agent 6. Then, however, the resulting configuration intersects the trap at the same slot and index as the previous configuration because the occurring change is immaterial for the intersection with the trap. Analogous reasoning applies for the execution of the loop transition of agent 5.

Clearly, if $O$ is a trap of an instance $\langle C_0, \vdash \rangle$, then we have $C' \cap O$ for every reachable configuration $C'$. Therefore, a set $I$ of traps induces an over-approximation of the set of reachable configurations, namely the set of configurations $C$ such that $C' \cap O$ for all $O \in I$. Algorithm 1 shows a CEGAR (counter-example guided abstraction refinement) loop, adapted from [15], to compute a set $I$ of traps proving a given safety property. Starting with $I = \emptyset$, the algorithm searches for a configuration (reachable or not) that intersects all traps of $I$ but violates the safety property. If there is none, the property holds, and the algorithm terminates; otherwise, the algorithm searches in Line 5 for a trap that is not intersected by the configuration, and adds it to $I$. It is well known that the set of such traps can be characterized as the solutions of a SAT formula.

4 Parameterized analysis

In the trap of Example 2 the slots $i.tlp$, for $i \in \{0, 1, 2, 4, 6\}$ are empty: all their entries are the empty set. Intuitively, this means that the trap expresses an invariant involving only the agents 3 and 5. The important fact is that this invariant is independent of how many other agents there are. Indeed, Figure 2 is just one instance of a family of traps for instances of this system. Each element of this family can be represented as a slotted word with a fixed number of slots, independent of the number of agents. For this we first remove all the empty slots. Observe, however, that this loses information because we no longer
know which are the indices of the agents of the two nonempty rows for the loop transitions. So, we also add a new slot \( \text{index} \) with alphabet \( \{p_0, p_1\} \) to mark these two agents. We call this process normalization.

The result of normalizing the family at the top of Figure 2 is shown in the middle of the figure. This finite expression can be seen as a parameterized invariant of the system. In the rest of the section we formalize the notion of a normalized trap, and present a simple abduction procedure that allows us to automatically generalize certain traps into regular expressions. The procedure can already be sketched in our running example. Consider, for example, the normalized slotted powerword \( \mathcal{N}_0 \ldots \mathcal{N}_{N-1} \) shown in the middle of Figure 2 for \( 2 < i \) and \( i + 2 < j < (N - 1) - 2 \). Note that Example 2 shows a conceptually similar trap, although there \( j = i + 2 \). Regardless, in Figure 2 we have \( \mathcal{N}_0 = \mathcal{N}_1 = \mathcal{N}_2 = \ldots = \mathcal{N}_{i-1} \) and \( \mathcal{N}_{i+1} = \mathcal{N}_{i+2} = \ldots = \mathcal{N}_{j-1} \). We prove that we can replace \( \mathcal{N}_0, \mathcal{N}_1, \mathcal{N}_2 \) by the regular expression \( \mathcal{N}_0^+ \mathcal{N}_1 \mathcal{N}_2 \), and similarly for \( \mathcal{N}_{i+1}, \mathcal{N}_{i+2}, \mathcal{N}_{i+3} \), with the guarantee that all words of the regular expression are traps of the corresponding instances. Indeed, we prove that any trap with 3-repetitions of the same letter allows for this generalization. The complete process of obtaining from an actual trap a normalized one and, then, a regular expression is sketched in the three lines of Figure 2.

### 4.1 Normalized traps and trap languages

We now explain how to define sets of traps similar to regular languages. This requires a finite alphabet which we obtain by only paying attention to the “relevant” iteration pointers. The following definition provides the technical details.

**Definition 6.** Let \( \mathcal{I} = \langle Q, q_0, I\text{Val}, T_{lo}, T_{lp} \rangle \) be a parameterized system. Let \( \mathcal{A} = \{p_0, \ldots, p_{m-1}\} \) be a finite set of agent names. A normalized trap of an instance \( \langle C_0, \vdash \rangle \) over the set of agent names \( \mathcal{A} \) is a slotted word \( \mathcal{N}_0, \ldots, \mathcal{N}_{N-1} \) over the set of slots

\[
\{\text{index}, \text{loc}, \text{Vars}, p_0, T_{lp}, \ldots, p_{m-1}, T_{lp}\}
\]

where the alphabet of index is \( \Sigma_{\text{index}} = \mathcal{A} \cup \{\square\} \), and the other alphabets are as for normal traps, satisfying the following conditions:

- For every name \( p_i \in \mathcal{A} \) there is exactly one letter \( \mathcal{N}_j \) with \( \mathcal{N}_j(\text{index}) = p_i \).
  
  We say that \( p_i \) is the name of the \( j \)-th agent.

- For every name \( p_i \in \mathcal{A} \) and for every loop transition \( t_j \in T_{lp} \) there is at least one \( 0 \leq k < N \) such that \( p_i.t_j(k) = \{\uparrow\} \).
The trap condition, as in Definition 5.

We let $\Sigma_{\mathcal{A}}$ denote the alphabet of normalized traps over $\mathcal{A}$. A trap language over $\Sigma_{\mathcal{A}}$ is a regular expression of the form $r_0 r_1 \ldots r_{i-1}$, where $r_i \in \{a, a^* \mid a \in \Sigma_{\mathcal{A}}\}$ for every $1 \leq i \leq \ell$.

The middle part of Figure 2 shows a normalized trap over $\mathcal{A} = \{p_0, p_1\}$. It is obtained by normalizing the trap at the top of the figure by means of the following procedure:

- Remove “empty” loop slots. Formally, if $i.t_j(k) = \emptyset$ for every $t_j \in T_{lp}$ and $0 \leq k < N$, remove all the slots of $i.T_{lp}$.

- Rename all remaining slots $0 \leq i_0, \ldots, i_{m-1} < N$ with the “abstract names” $\{p_0, \ldots, p_{m-1}\}$. Intuitively: $p_k$ is a placeholder for some index of the word. The corresponding concrete index is marked via the additional slot index.

- Use the index slot to give the letters $N_0, \ldots, N_{m-1}$ the names $p_0, \ldots, p_{m-1}$. Label the others with $\square$.

The bottom of the figure shows the trap language obtained by normalizing all the traps shown at the top for all possible values of the indices $i$ and $j$. The next theorem is the basis of the abduction process that, given a normalized trap of one instance of the system, produces, if possible, an infinite trap language of normalized traps.

**Theorem 1.** Let $\mathcal{I}$ be a parameterized system. Let $N_0, \ldots, N_{N-1} \in \Sigma_{\mathcal{A}}$ be a normalized trap of the instance of $\mathcal{I}$ with $N$ agents. If $N_i(index) = \square$ and $N_i = N_{i+1} = N_{i+2}$, then for every $k \geq 1$ the word

$$N_0 \cdots N_{i-1} \cdot N_i^k \cdot N_{i+1} \cdots N_{N-1}$$

is a normalized trap of the instance of $\mathcal{I}$ with $N + k$ agents.
Before we prove this theorem, we introduce the notions $\text{move}_k^{[-]}$, $\text{move}_k^{[-][-]}$, $\text{drop}_k$. Conceptually, $\text{move}_k^{[-]}$ and $\text{move}_k^{[-][-]}$ move pointers of the loop transitions $T \subseteq T_{lp}$, which are currently pointing to $k$, to $k-1$ and $k+1$ respectively. For example consider

$$\text{move}_1^{\{n_p\}[-]} \left( \begin{array}{ccc}
\text{break} & t_p & t_p \\
\uparrow & \uparrow & \uparrow
\end{array} \right) = \left( \begin{array}{ccc}
\text{break} & t_p & t_p \\
\uparrow & \uparrow & \uparrow
\end{array} \right).$$

and, analogously,

$$\text{move}_1^{\{n_p\}[-]} \left( \begin{array}{ccc}
\text{break} & t_p & t_p \\
\uparrow & \uparrow & \uparrow
\end{array} \right) = \left( \begin{array}{ccc}
\text{break} & t_p & t_p \\
\uparrow & \uparrow & \uparrow
\end{array} \right).$$

Note that $\text{move}_0^{T[-]}$ is not well-defined and neither is $\text{move}_k^{T[-]}$ if $k$ is the last index of a configuration since there is no index to move the transition pointers to.

Formally, we say $\text{move}_k^{T[-]}(c_0 \ldots c_{n-1}) = c_0' \ldots c_{n-1}'$ such that $c'_\ell = c_\ell$ for all $\ell \in [n] \setminus \{k-1,k\}$. The values of variables, the current location and loop transitions that are not “moved” do not change: $c'_{k-1}(s) = c_{k-1}(s)$ and $c'_s(s) = c_s(s)$ for $s \in \text{Vars} \cup \{\text{loc}\} \cup \{i.t: t \in T_{lp} \setminus T\}$ and $i \in [n]$. Pointers in $T$, however, move to the left (or to the right if we consider $\text{move}_k^{T[-][-]}$ instead): $c'_\ell(\ell'.t) = \square$ while $c'_{k-1}(\ell'.t) = \uparrow$ if either $c_{k-1}(\ell'.t) = \uparrow$ or $c_{k}(\ell'.t) = \uparrow$ and otherwise $c'_{k-1}(\ell'.t) = \square$ for all $\ell' \in [n]$ and $t \in T$.

Secondly, $\text{drop}_k$ describes how to remove a specific index $k$ from a configuration. Essentially, we remove all slots $k.T_{lp}$ and the column $k$. As an example, consider

$$\text{drop}_3 \left( \begin{array}{ccc}
\text{break} & t_p & t_p \\
\uparrow & \uparrow & \uparrow
\end{array} \right) = \left( \begin{array}{ccc}
\text{break} & t_p \\
\uparrow & \uparrow
\end{array} \right).$$

Hence, $\text{drop}_k(c_0 \ldots c_{n-1}) = c_0' \ldots c_{n-2}'$ such that

- for $\ell_1 < k$, $\ell_2 < k$, $t \in T_{lp}$ we have $c'_{\ell_1}(\ell_2.t) = c_{\ell_1}(\ell_2.t)$ and $c'_{\ell_1}(s) = c_{\ell_1}(s)$ for all $s \in \text{Vars} \cup \{\text{loc}\}$,
- for $\ell_1 < k$, $k \leq \ell_2$, $t \in T_{lp}$ we have $c'_{\ell_1}(\ell_2.t) = c_{\ell_1}(\ell_2.t)$,
- for $k \leq \ell_1$, $\ell_2 < k$, $t \in T_{lp}$ we have $c'_{\ell_1}(\ell_2.t) = c_{\ell_1+1}(\ell_2.t)$ and $c'_{\ell_1}(s) = c_{\ell_1+1}(s)$ for all $s \in \text{Vars} \cup \{\text{loc}\}$, and
- for $k \leq \ell_1$, $k \leq \ell_2$, $t \in T_{lp}$ we have $c'_{\ell_1}(\ell_2.t) = c_{\ell_1+1}(\ell_2+1.t)$.

We make a few observations about $\text{drop}_k$:

- $\text{drop}_k$ yields a configuration if no slot of any loop transition of agent $k$ contains the value $\uparrow$.
- Let $C$ be a configuration and $O$ a compatible powerword. If $c_k(s) \notin O_k(s)$ for all slots $s$ and $c_k(\ell.t) \notin O_k(\ell.t)$ for all $\ell$ and $t \in T_{lp}$ then $\text{drop}_k(C) \cap \text{drop}_k(O)$ iff $C \cap O$.

Equipped with these notions we prove Theorem 1.
Proof. For the sake of contradiction assume that the statement of Theorem 11 is incorrect. Then, we can fix a minimal $k_0$ such that
the normalized word $A_0 \ldots A_{i-1} A_i \ldots A_{i+k_0} A_{i+1} \ldots A_{N-1}$ gives a powerword $O$ which is not a trap in the instance of $N + k_0 - 1$ agents.

Therefore, let $C = c_0 \ldots c_{N-k_0-2}$ and $C' = c'_0 \ldots c'_{N-k_0-2}$ be configurations of the instance with $N + k - 1$ agents such that $C \vdash C'$ and $C \not\models O$, but $C' \not\models O$.

Now, observe that $k_0 > 1$ since $k_0 = 1$ immediately contradicts with the prerequisites of the theorem. However, since $k_0$ is minimal, $A' = A_0 \ldots A_{i-1} A_i \ldots A_{i+k_0-1} A_{i+1} \ldots A_{N-1}$ is a normalized trap of $N + k - 2$ agents. Let $O'$ be the corresponding powerword – which is, in fact, a trap. Note that $O' = \text{drop}_i(O) = \text{drop}_{i+1}(O) = \ldots = \text{drop}_{i+k_0}(O)$.

Consider the case where $C \vdash C'$ is an instance of a local transition $t$. Thus, $C$ and $C'$ differ at exactly one index $j$ and there at most in slots from $\text{Vars} \cup \{\text{loc}\}$. Pick now $m \in \{i, i+1, i+2, i+3\} \setminus \{j\}$. W.l.o.g. we assume that $m - 1 \in \{i, i+1, i+2, i+3\}$ (otherwise exchange $m - 1$ with $m + 1$ and $\text{move}_m^{T_{ip}[\cdot]}$ with $\text{move}_m^{T_{ip}[\cdot]}$ within the following argument). By the definition of instances of local transitions it is straightforward to see that $\text{move}_m^{T_{ip}[\cdot]}(C) \not\models \text{move}_m^{T_{ip}[\cdot]}(C)$. However, we still have $\text{move}_m^{T_{ip}[\cdot]}(C) \cap O$ but $\text{move}_m^{T_{ip}[\cdot]}(C') \not\models O$ since $O_{m-1} = O_m$ and, thus, $O_{m-1}(\ell, t) = O_m(\ell, t)$ for all $\ell \in [n + k_0 - 1]$ and $t \in T_{ip}$. Since agent $m$ does not contain any $\uparrow$ values anymore ($\text{drop}_m \circ \text{move}_m^{T_{ip}[\cdot]}(C)$) and $\text{move}_m \circ \text{move}_m^{T_{ip}[\cdot]}(C)$ indeed are configurations.

Moreover, by instantiating the same local transition for agent $j$ or, if $m \leq j$, $j - 1$ gives ($\text{drop}_m \circ \text{move}_m^{T_{ip}[\cdot]}(C) \not\models (\text{drop}_m \circ \text{move}_m^{T_{ip}[\cdot]}(C'))$). Note that $O_t(m, t) = \emptyset$ for all $t \in T_{ip}$ since $A_m(\text{index}) = \emptyset$ and $c_m(s) \notin O_m(s)$ for all slots $s$ because $j \neq m$ and $C' \not\models O$. Thus, by our observations about $\text{drop}_m$, we have ($\text{drop}_m \circ \text{move}_m^{T_{ip}[\cdot]}(C) \cap O'$ because $O' = \text{drop}_m(O)$, but $\text{drop}_m \circ \text{move}_m^{T_{ip}[\cdot]}(C') \not\models O'$ contradicting that $O'$ is a trap.

Consider the case that $C \vdash C'$ is an instance of a loop transition $t$. Then there are indices $j$ and $p$ such that $j$ is the agent executing the loop transition while $p$ is the agent that $j$ currently inspects. If the loop transition fails, $p$ changes from $C$ to $C'$ only in slot $j$ from $\uparrow$ to $\emptyset$ while $j$ changes from $C$ to $C'$ only in slot $\text{loc}$ from $t$ to $\text{target}_{\text{fail}}$. In this case pick $m \in \{i, i+1, i+2, i+3\} \setminus \{j, p\}$. Again, we assume that $m - 1 \in \{i, i+1, i+2, i+3\}$. Using $O_i = O_{i+1} = O_{i+2} = O_{i+3}$ and, $m$ not being involved in this instance of $t$ allows us to consider $D = \text{move}_m^{T_{ip}[\cdot]}(C) \vdash \text{move}_m^{T_{ip}[\cdot]}(C') = D'$ instead. Exploiting the same principles above – namely, the fact that the $m, t$ slot is empty for every letter in $O$ and the agent $m$ not changing from $D$ to $D'$ – allows us to deduce that $\text{drop}_m(D) \cap O'$ while $\text{drop}_m(D') \not\models O'$ which contradicts the assumption of $k_0$ being minimal.

It remains to consider the case that this transition is an instance of $t$ for inspector $j$ and inspectee $p$ which is successful. If $p$ is the last agent, then $C$ and $C'$ only differ at indices $j$ and $p$. Hence, the pattern of the before mentioned cases applies again.

Otherwise, only agents $j$, $p$, and $p + 1$ change. Then, however, there is $m \in \{i, i+1, i+2, i+3\} \setminus \{j, p, p+1\}$. Repeating the same arguments as above we can show that $O'$ cannot be a trap since ($\text{drop}_m \circ \text{move}_m^{T_{ip}[\cdot]}(C) \vdash (\text{drop}_m \circ \text{move}_m^{T_{ip}[\cdot]}(C'))$ is an instance of a transition for the instance of size $N + k_0 - 2$.

\[\square\]
4.2 Proving safety properties

Recall that in [19] we consider parameterized systems without loop transitions, i.e., having only local transitions and the transitions described in Remark 1. For these systems, the problem whether every global configuration satisfying a family of trap languages also satisfies a desired safety property can be reduced to the satisfiability problem for WS1S, which is decidable (see also [9, 10]). Unfortunately, this is no longer the case for systems with loop transitions.

Let \( \mathcal{I} = \langle Q, q_0, \text{Val}, \mathbb{T}_{lo}, \mathbb{T}_{lp} \rangle \) be a parameterized system. For every configuration \( C \), let \( \mathcal{E}_q \) denote the set of global configurations of all instances of \( \mathcal{I} \) such that at least one agent is in state \( q \). Similarly, for every variable \( x \) and every value \( v \) of \( \text{Values}_x \), let \( \mathcal{E}_{v,x} \) be the set of global configurations of all instances of \( \mathcal{I} \) such that for at least one agent the variable \( x \) has value \( v \). A safety property \( \mathcal{P}(C) \) is a Boolean combination of the predicates \( C \in \mathcal{E}_q \) and \( C \in \mathcal{E}_{v,x} \). An example safety property could be “there is no agent in the state critical, or there is an agent in state loop and no agent such that \( b = \top \)”. Let \( \mathcal{L} = \{ \mathcal{L}_1, \ldots, \mathcal{L}_k \} \) be a finite set of trap languages. Abusing language, let \( \mathcal{L}(C) \) denote the predicate that holds for a global configuration \( C \) if \( C \cap O \) for every trap \( O \in \mathcal{L} \) compatible with \( C \).

Given \( \mathcal{I}, \mathcal{P}, \mathcal{L} \), we want to decide if every configuration \( C \) satisfying \( \mathcal{L}(C) \) also satisfies \( \mathcal{P}(C) \), i.e., whether the invariants of \( \mathcal{L} \) are sufficient to prove the safety property \( \mathcal{P} \). Since \( \mathcal{L}(C) \) is an inductive predicate (i.e., \( \mathcal{L}(C) \) and \( C \models C' \) imply \( \mathcal{L}(C') \)), this is the case if (1) \( \mathcal{P} \) holds for every initial configuration of every instance of \( \mathcal{I} \), and (2) \( \mathcal{L}(C) \wedge \mathcal{P}(C) \wedge C \models C' \) implies \( \mathcal{P}(C') \). This leads to the following definition:

**Definition 7.** Let \( \mathcal{I}, \mathcal{L}, \mathcal{P} \) be a parameterized system, a trap language, and a safety property, respectively. The inductivity problem consists of deciding if \( \mathcal{L}(C) \wedge \mathcal{P}(C) \wedge C \models C' \) holds for every two configurations \( C, C' \) of \( \mathcal{I} \).

**Theorem 2.** The inductivity problem is undecidable.

*Sketch.* Any loop transition \( t \) implicitly provides a reference from agent to agent. We can use both this and the order of agents to enforce grid structures within instances. This allows us to give a reduction of the problem whether a periodic tiling for a given set of Wang tiles exists to the question above. Then, the result follows from the undecidability of finding periodic tilings for Wang tiles [27]. For the formal details refer to Appendix B of the full version [17].

Due to this result, there is no hope of reducing the safety problem for systems with loop transitions to the satisfiability problem of a decidable logic. It is still possible, though, to reduce it to the satisfiability problem of first-order logic with monadic predicates and one function symbol, and run it through an automatic first-order theorem prover. Informally, we must embed the statement

\[
\forall C, C', (\mathcal{L}(C) \wedge \mathcal{P}(C) \wedge C \models C') \rightarrow \mathcal{P}(C')
\]

in first-order logic.

**Embedding into first-order logic.** In [19] we consider configurations that are solely defined by the values of the local variables and the state of a process. This can be encoded by a set of monadic second-order variables; namely, we introduce a monadic second-order variable \( X^{var}_{val} \) for all values \( val \) of all \( var \in \text{Vars} \).

In the presence of loop transitions, the local configuration also includes the pointer, whose value ranges from 0 to \( N - 1 \), where \( N \) is the number of processes. For this we introduce one function symbol \( f \).
This function symbol is interpreted by maps from \{0, \ldots, N-1\} to \{0, \ldots, N-1\}. Combining this, we describe a configuration \(C\) as monadic predicates \(X^\text{var}_{\text{val}}\) for every \(\text{val} \in \text{Values}_{\text{var}}\) for all \(\text{var} \in \text{Vars}, X_q\) for every \(q \in Q\), \(X_t\) for every \(t \in \mathbb{T}_{tp}\) and a unary function symbol \(f\). It is obvious, however, that not every interpretation of these monadic predicates and the function symbol \(f\) properly describes a configuration. Only models which ensure that every variable is currently in exactly one value for every agent, and that every agent is currently in exactly one location can be considered proper. For this, we introduce the formula \(\eta\) which ensures these properties. Namely, we define

\[
\eta = \forall i: \left( \bigwedge_{\text{var} \in \text{Vars}} \bigvee_{\text{val} \in \text{Values}} X^\text{var}_{\text{val}}(i) \land \bigwedge_{\text{val}' \in \text{Values} \setminus \{\text{val}\}} \neg X^\text{var}_{\text{val}'}(i) \right).
\]

\(\eta\) enforces that every agent currently is in exactly one location and sets every variable to exactly one value. We introduce notions which leverage this observation to increase readability of formulas. Namely, we write \(\text{var}(i) = \text{val}\) and \(\text{loc}(i) = q (\text{loc}(i) = t)\) to express \(X^\text{var}_{\text{val}}(i)\) and \(X_q(i) (X_t(i))\) respectively. Moreover, we introduce \(\text{var}(i) = \text{var}(j)\) as a short form of \(\bigwedge_{\text{val} \in \text{Values}_{\text{val}}} X^\text{var}_{\text{val}}(i) \leftrightarrow X^\text{var}_{\text{val}}(j)\) and, as its dual, \(\neg \text{var}(i) \neq \text{var}(j)\) expresses \(\neg (\text{var}(i) = \text{var}(j))\).

To encode the inductivity problem it is necessary to relate two configurations \(C\) and \(C'\) such that \(C \vdash C'\). For this, we introduce a primed version of all monadic predicates and the function symbol \(f\) which we use to encode a configuration to represent \(C'\). Additionally, we axiomatize a total linear order \(\leq\) between the constant symbols 0 and \((N-1)\) in a formula \(\psi\). Since \(\psi\)’s definition is fairly standard we omit it here for brevity. Without increasing the expressiveness of the theory of this total linear order, we introduce \(t < t'\) to express \(t \neq t' \land t \leq t'\) and \(t + 1\) as a term to refer to the immediate successor of \(t\) w.r.t. \(\leq\) (this requires that \(t\) cannot be interpreted with \(N - 1\) which is ensured for all occurrences). As \(\psi\), the encoding of \(C \vdash C'\) is now fairly standard. We postulate the existence of a formula \(\tau\) which relates the predicates and the function symbols \(f\) and \(f'\) of \(C\) and \(C'\). One clause in \(\tau\), for example, which encodes the loop transition \(\mathbb{T}_{tp}\) of Example\[\text{II}\] (cp. Figure\[\text{II}\]) is

\[
\forall \ell: (\text{var}(\ell) = \text{var}'(\ell))
\land \exists i: \text{loc}(i) = t
\land \forall k \neq i: (\text{loc}(k) = \text{loc}'(k) \land f(k) = f'(k))
\land \left( (f(i) \neq i \land (f(i)) = \top) \rightarrow \text{loc}'(i) = \text{break} 
\land (f(i) = i \land f(i)) = \bot \land i < (N - 1) \rightarrow \text{loc}'(i) = t \land f'(i) = f(i) + 1 \right).
\]

Secondly, the inductivity problem relies on only considering configurations \(C\) which intersect all compatible normalized traps of the computed trap languages. The trap language at the bottom of Figure\[\text{II}\]describes one of these trap languages. Careful analysis of Example\[\text{II}\], however, reveals a stronger invariant\[\text{II}\]. Namely, that all words in

\[
\begin{array}{cccc}
\text{index} & \text{loc} & \text{var} & \text{p}_0 \text{–} \text{p}_1 \\
\square & \emptyset & \emptyset & \{\text{break,loop}\} \\
\circ & \emptyset & \emptyset & \{\text{break,loop}\} \\
\{\uparrow\} & \{\uparrow\} & \{\uparrow\} & \{\uparrow\} \\
\{\uparrow\} & \{\uparrow\} & \{\uparrow\} & \{\uparrow\} \\
\end{array}
\]

\[\text{(3)}\]

\[2\text{In fact, this is one of the invariant languages that our tool } \text{heron} \text{ computes for this example.}\]
are normalized traps in instances of the corresponding size. On this basis, we can derive a formula \( \Phi = \eta \land \psi \land \varphi \) whose models are the global configurations \( C \) that intersect all traps of this language, and are compatible with \( C \). We already discussed the nature of \( \eta \) and \( \psi \). It remains to introduce \( \varphi \) which restricts models of \( \Phi \) to configurations which intersect every compatible trap of the language. The traps of the language are characterized by the positions of the abstract names \( p_0 \) and \( p_1 \). So \( \varphi \) is of the form “for every two indices \( 0 \leq p_0 < p_1 < N \), the (unique) trap of the language corresponding to these values of \( p_0 \) and \( p_1 \) intersects \( C \)”. The formula splits into several cases corresponding to having an intersection of \( C \) and the considered trap at some index \( i < p_0 \), at \( p_0 \), between \( p_0 \) and \( p_1 \), or at \( p_1 \). Consider the case where an intersection occurs at some index \( i < p_0 \): since the letters to the left of the letter with index \( p_0 \) are of the form \((\Box, 0, \emptyset, \{\uparrow\}, \{\downarrow\})\), this is the case if one of the agents \( p_0 \) or \( p_1 \) is currently executing its loop transition, and its pointer points to \( i \). This is captured by the formula \( t(p_0) = i \) or \( t(p_1) = i \). Proceeding like this for the other cases gives

\[
\varphi = \forall p_0 < p_1 : \begin{cases} \\
\exists i < p_0 \cdot t(p_0) = i \lor t(p_1) = i \\
\lor \forall \forall p_0 < i < p_1: t(p_0) = i \\
\lor \forall \forall p_0 < i < p_1: \lor t(p_0) = p_0 \lor t(p_1) = p_0 \\
\lor \forall \forall p_0 < i < p_1: \lor t(p_1) = i \\
\lor \forall \forall p_0 < i < p_1: \lor t(p_0) = p_0 \\
\lor \forall \forall p_0 < i < p_1: \lor t(p_1) = p_1 \\
\lor \forall \forall p_0 < i < p_1: \lor t(p_0) = p_1 \\
\lor \forall \forall p_0 < i < p_1: \lor t(p_1) = p_1
\end{cases}.
\]

Recall from Definition 6 that trap languages are defined as regular expressions of the form \( r_0 r_1 \ldots r_{\ell-1} \) where each \( r_i \) is either a single letter; i.e., some \( a \), or the arbitrary repetition of a single letter; i.e., the expression \( a^* \). Hence, it is straightforward to define \( \varphi \) for a finite collection of trap languages by generalizing the example above. Again, we avoid giving this definition for brevity.

The inductiveness problem corresponds now to the question whether

\[
(\Phi \land \tau \land \mathcal{P}) \rightarrow \mathcal{P}'
\]

is valid where \( \mathcal{P} \) and \( \mathcal{P}' \) are formalizations of the desired safety property; e.g., \( \neg(\exists i \neq j : \text{loc}(i) = \text{critical} \land \text{loc}(j) \neq \text{critical}) \) and \( \neg(\exists i \neq j : \text{loc}'(i) = \text{critical} \land \text{loc}'(j) \neq \text{critical}) \) for Example 1.

5 Experimental results

We implemented the approach we describe in this paper in our tool heron \([18, 35]\) which is written in the Python programming language. We implemented Algorithm 1 using clingo \([22]\) and relied for first-order theorem proving on VAMPIRE \([29]\) and CVC4 \([6]\). Essentially, heron runs Algorithm 1 and normalizes the occurring traps. In these normalized traps, it identifies repetitions which can be safely generalized via Theorem 1. Additionally, we try to obtain succinct invariants. That is, languages with small regular expressions. As an example, we consider the language from Equation (3) as succinct. To this end, once heron encounters a repetition of letters in a normalized traps it proceeds to shrink and expand this repetition and checks whether the resulting normalized traps correspond to actual traps. If a repetition can be expanded to the generalization threshold it can be safely repeated arbitrarily often. This heuristic is relatively cheap since it simply relies on checking whether a normalized trap is indeed a trap but allows us to easily compute succinct but general trap languages.

Examples As a benchmark we used classical examples of mutual exclusion algorithms. Namely, the mutual exclusion algorithm of Dijkstra \([12]\) (here we prove the reduced version as described in Example 1).
as well as a precise formulation), Knuth [28], de Bruijn [11], and Eisenberg & McGuire [13]. For all these examples we can prove fully automatically that they ensure the mutual exclusion property for their respective critical sections. To the best of our knowledge these are the first fully automatic proofs for the algorithms of Knuth, de Bruijn, and Eisenberg & McGuire for a model with non-atomic global checks.

However, all but Example 1 require the use of global pointers to processes; i.e., we introduce a variable with some value in \([N]\) for every instance of size \(N\) and processes can check the current value of local variables of the process this pointer currently refers to, set their own identifier as the current value of the pointer, or upon encountering a certain state in an agent during an iteration setting the pointer to this agent. We adapt our approach to account for this: for every pointer we introduce an additional slot with a corresponding alphabet of \(\{↑, □\}\) and maintain the invariant that every configuration contains exactly one agent for which a pointer slot holds the value \(↑\). This makes it necessary to broaden the scope of Theorem 1. Essentially, we obtain now that every transition might interact with all agents a pointer currently points to. Careful examination of the proof of Theorem 1 suggests that we mainly rely on the fact that for every transition only a finite amount of agents are essential. Adding pointer variables might increase the amount of agents that are considered by a transition. This is immaterial for the spirit of Theorem 1 though, but increases the amount of repetitions necessary to allow for safe generalization. Fortunately, as long as the necessary invariants happen to be simple, the direct performance impact is limited to checking a few more easily obtained traps for finite instances. In Appendix C of the full version [17] we shortly discuss the formalization of this observation.

**Results** The details of our experiments can be found in Figure 3. We report in the first column the name of the considered algorithm. The second column contains the time it took to prove the mutual exclusion property for this system. In the third column we give the maximal size for which we instantiated and analyzed the system. The fourth column reports the amount of traps that were computed for the various instances of the system while the fifth column gives the amount of trap languages that were actually necessary to establish the mutual exclusion property. heron axiomatizes for the inductivity problem a minimal size of the considered configurations which exceeds the already analyzed instances. This axiom, however, is not used for the proof of all examples; i.e., all inductivity queries can be solved without this axiom in comparable time. This means that the reported trap languages indeed are sufficient to establish the mutual exclusion property for all instances. In the sixth column we report the maximum amount of abstract indices that occur in any of the trap languages. Finally, the last column contains the maximum time it took to solve the final first-order problems for the various transitions. We observe that the proof search does not dominate the running time, the real bottleneck being the analysis of the finite instances.

The information of Figure 3 are extracted from log files that heron produces on its execution. We want to stress specifically that heron does not only automatically prove the desired property but also
gives a detailed and human-readable explanation. This explanation presents itself in form of the computed traps for considered finite instances as well as the computed trap languages which suffice to prove the inductivity problem for all instances which are larger than the analyzed finite instances. These trap languages can be read and understood by humans, and are for all considered examples sufficient to prove inductivity of the mutual exclusion property for all instances of the parameterized system. As a complete example, consider the language in Equation (3) and

\[
\text{index}
\]

\[
\text{loc}
\]

\[
\text{var}
\]

\[
\begin{align*}
\left(\square \right)^* \left( \emptyset \right) \left( \{\text{initial}\} \right) \left( \top \right) \left( \emptyset \right) \left( \emptyset \right) \left( \emptyset \right) \left( \emptyset \right) \left( \emptyset \right) \left( \emptyset \right) \\
\end{align*}
\]

which are the only languages \textsf{heron} computes to prove Example 1. Additionally, \textsf{heron} gives a series of first-order problems in the widely adopted TPTP format [34] which describe the inductiveness checks of the proven property. This allows a user to leverage the advanced tool support for first-order theorem proving [6, 25, 33] to verify the proof and, simultaneously, better understand the analyzed system.

In fact, the data of Figure 3 shows that mutual exclusiveness in the critical section can be established with very few invariants (e.g., in less than 7 trap languages). Moreover, all these languages of normalized traps need at most two references to indices. Hence, the invariants express—in some sense—local properties. Finally, we want to draw attention to the fact that first-order theorem proving is extremely efficient once all necessary invariants are established. This suggests that the presented approach is practically viable despite Theorem 2.

6 Conclusion

In our previous work, we showed the value of structural invariants of parameterized systems for their analysis [9]. This contribution is the natural expansion of our abduction principles from [19] to parameterized systems with non-atomic global checks. Although one sacrifices the decidability of the inductivity problem in this expansion, the experimental data suggests that it remains practically viable. This contrasts observations from [9, 19] where the inductivity problem is decidable albeit it often fails in practice due to its computational complexity. Moreover, most of the time is spent on the analysis of the finite instances, while generalisations of suitable finite invariants can be verified very quickly. We therefore conclude that this expansion is worthy of consideration and, in a broader view, establishes analysis of parameterized systems via abduction of structural invariants as a promising direction. Moreover, we believe that our work complements existing approaches like view abstraction [2]. Especially, since the invariants \textsf{heron} computes for Dijkstra’s algorithm can be similarly expressed in the formalism of view abstraction. However, \textsf{heron} obtains these languages from structural analysis of instances while in view abstraction one abstracts from the reachable set of configurations for fixed sizes. Also, \textsf{heron} computes comparatively very few invariants but needs significantly more time to do so. Thus, future work can focus on improving the analysis of finite instances via structural properties. Moreover, it is tempting to further explore the trade-offs between how strong an analysis is and how easy the considered structural properties can be abduced to the general case. In particular, a richer set of considered invariants should allow more diverse communication structures between the agents. Additionally, one can consider applying abduction principles to proofs of liveness properties in, e.g., Petri nets [16].
References

[1] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson & Yih-Kuen Tsay (1996): General Decidability Theorems for Infinite-State Systems. In: LICS, IEEE Computer Society, pp. 313–321, doi:10.1109/LICS.1996.561359.

[2] Parosh Aziz Abdulla, Frédéric Haziza & Lukás Holík (2016): Parameterized verification through view abstraction. Int. J. Softw. Tools Technol. Transf. 18(5), pp. 495–516, doi:10.1007/s10009-015-0406-x.

[3] Parosh Aziz Abdulla, Noomene Ben Henda, Giorgio Delzanno & Ahmed Rezine (2008): Handling Parameterized Systems with Non-atomic Global Conditions. In: VMCAI, Lecture Notes in Computer Science 4905, Springer, pp. 22–36, doi:10.1007/978-3-540-78163-9_7.

[4] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson & Mayank Saksena (2004): A Survey of Regular Model Checking. In: CONCUR, LNCS 3170, Springer, pp. 35–48, doi:10.1007/978-3-540-28644-8_3.

[5] Parosh Aziz Abdulla, A. Prasad Sistla & Muralidhar Talupur (2018): Model Checking Parameterized Systems. In: Handbook of Model Checking, Springer, pp. 685–725, doi:10.1007/978-3-319-10575-8_21.

[6] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds & Cesare Tinelli (2011): CVC4. In: CAV, Lecture Notes in Computer Science 6806, Springer, pp. 171–177, doi:10.1007/978-3-642-22110-1_14.

[7] Kai Baukus, Saddek Bensalem, Yassine Lakhnech & Karsten Stahl (2000): Abstracting WS1S Systems to Verify Parameterized Networks. In: TACAS, LNCS 1785, Springer, pp. 188–246, doi:10.1007/978-3-319-045190-5_13.

[8] Michael Blondin, Javier Esparza, Martin Helfrich, Antonín Kucera & Philipp J. Meyer (2020): Checking Qualitative Liveness Properties of Replicated Systems with Stochastic Scheduling. In: CAV (2), LNCS 12225, Springer, pp. 372–397, doi:10.1007/978-3-030-53291-8_20.

[9] Marius Bozga, Javier Esparza, Radu Iosif, Joseph Sifakis & Christoph Welzel (2020): Structural Invariants for the Verification of Systems with Parameterized Architectures. In: TACAS (1), Lecture Notes in Computer Science 12078, Springer, pp. 228–246, doi:10.1007/978-3-030-45190-5_13.

[10] Marius Bozga, Radu Iosif & Joseph Sifakis (2021): Checking deadlock-freedom of parametric component-based systems. J. Log. Algebraic Methods Program. 119, p. 100621, doi:10.1016/j.jlamp.2020.100621.

[11] N. G. de Bruijn (1967): Additional comments on a problem in concurrent programming control. Commun. ACM 10(3), pp. 137–138, doi:10.1145/363162.363167.

[12] Edsger W. Dijkstra (2002): Cooperating Sequential Processes, pp. 65–138. Springer New York, New York, NY, doi:10.1007/978-1-4757-3472-0_2.

[13] Murray A. Eisenberg & Michael R. McGuire (1972): Further Comments on Dijkstra’s Concurrent Programming Control Problem. Commun. ACM 15(11), p. 999, doi:10.1145/355606.361895.

[14] Javier Esparza, Pierre Ganty, Jérôme Leroux & Rupak Majumdar (2017): Verification of population protocols. Acta Informatica 54(2), pp. 191–215, doi:10.1007/s00236-016-0272-3.

[15] Javier Esparza, Ruslán Ledesma-Garza, Rupak Majumdar, Philipp J. Meyer & Filip Niksic (2014): An SMT-Based Approach to Coverability Analysis. In: CAV, Lecture Notes in Computer Science 8559, Springer, pp. 603–619, doi:10.1007/978-3-319-08867-9_40.

[16] Javier Esparza & Philipp J. Meyer (2015): An SMT-based Approach to Fair Termination Analysis. In: FM-CAD, IEEE, pp. 49–56, doi:10.1109/FM-CAD.2015.7542252.

[17] Javier Esparza, Mikhail Raskin & Christoph Welzel (2021): Abduction of trap invariants in parameterized systems. Available at https://arxiv.org/abs/2108.09101.

[18] Javier Esparza, Mikhail Raskin & Christoph Welzel (2021): heron, git repository. https://gitlab.lrz.de/i7/heron.
A Formal definition of the semantics

Here we describe the formal definition of the semantics of instances in our parameterized system. For this, let \( c = c_0 \ldots c_{n-1} \) and \( c' = c'_0 \ldots c'_{n-1} \) be configurations. Additionally, we write \( c_i[s \mapsto v] \) for \( s \in S_I \) and \( v \in \Sigma_s \) to denote a letter equivalent to \( c_i \) but where slot \( s \) now contains the value \( v \).

Local transition Let \( t \) be a local transition as in Eq. (1). Now fix any \( i \in [n] \) such that \( C_i(\text{loc}) = \text{origin} \). Then \( c \vdash c' \) if \( c_j = c'_j \) for all \( j \in [n] \setminus \{i\} \) and \( c'_i = c_i[s_1 \mapsto v_1] \ldots [s_k \mapsto v_k][\text{loc} \mapsto \text{target}] \).
Iterating transition Let $t$ be an iterating transition as in Eq. (2). Again, fix any index $i \in [n]$. We distinguish a few different cases:

- If $c_i(loc) = origin$ then agent $i$ just starts the iteration of $t$. Consequently, we say $c \vdash c'$ if $c'_0 = c_0[t.i \mapsto \top]$, $c'_j = c_j[t.i \mapsto \Box]$ for all $j \in [n] \setminus 0$, and $c' = c[i][loc \mapsto t]$. Note, that for the special case that $i = 0$ we require $c_0 = c_0[t.i \mapsto \top][loc \mapsto t]$ and $c'_j = c_j$ for all $j \in [n] \setminus \{0\}$ instead.

- If $c_i(loc) = t$ then agent $i$ already executes the transition $t$. Thus, there exists exactly one index $j$ such that $c_j(t.i) = \top$. We distinguish cases on the value of $j$:
  
  - If $j = n - 1$ then agent $i$ changes its location. Either to $target_{\text{suc}}$ or $target_{\text{fail}}$; depending on whether agent $j$ satisfies the condition $\varphi$. Hence, $c'_j = c_j[loc \mapsto target_{\text{suc}}]$ if $c_j = \top$. Additionally, $c'_j = c_j[t.i \mapsto \Box]$. Note that, if $j = i$, then predicates $\text{self}$ are interpreted as true otherwise they are interpreted as false. In any way, $c'_k = c'_k$ for all $k \in [n] \setminus \{i, j\}$. Analogously, we have $c'_i = c_i[loc \mapsto target_{\text{fail}}]$ and $c'_j = c_j[t.i \mapsto \Box]$ if $c_j \neq \top$.
  
  - If $i \neq j < n - 1$ then we distinguish whether agent $j$ satisfies the condition $\varphi$: if $c_j = \top$, then $c'_j = c_j[t.i \mapsto \Box]$ and $c_{j+1}[t.i \mapsto \top]$ while $c'_k = c_k$ for all $k \in [n] \setminus \{j, j + 1\}$. On the other hand; i.e., if $c_j \neq \top$, we have $c'_i = c_i[loc \mapsto target_{\text{fail}}]$ and $c'_j = c_j[t.i \mapsto \Box]$ while $c'_k = c_k$ for all $k \in [n] \setminus \{i, j\}$.

B Proof of Theorem 2

Theorem 2. The inductivity problem is undecidable.

Proof. The proof relies on a reduction from existence of a periodic tiling, using agent adjacency and iteration progress to model two directions on the grid.

Definition 8 (Periodic tiling of Wang tiles). Let $C$ be a finite set of colors. A Wang tiles is a mapping $\tau: \{N,E,S,W\} \rightarrow C$. A finite set of Wang tiles is called a tile set.

A periodic tiling of a tile set $\mathbb{T}$ is a function $f: [n] \times [m] \rightarrow \mathbb{T}$ for some $n, m$ such that $f(i, j)(E) = f(i, j \oplus_m 1)(W)$ and $f(i, j)(N) = f(i \oplus_n 1, j)(S)$.

We choose $\mathcal{S}$ to have the variables $\text{Vars} = \{\text{badflag}, \text{firstRow}, \text{firstCol}, \text{lastRow}, \text{lastCol}, \text{tile}\}$ where badflag, firstRow, firstCol, lastRow, and lastCol are Boolean variables; i.e., the values for these variables are $\top$ and $\bot$. Finally, we set $\text{Values}_{\text{tiles}}$ to $\mathbb{T}$.

Now, we consider $\mathcal{S}$ of two transitions: one loop transition $t = (q_1, \bot, q_{\bot}, q_{\bot})$ where the condition $\bot$ is a logically unsatisfiable statement; e.g., badflag $= \top \land$ badflag $= \bot$, and one local transition $(q_j, \text{badflag} = \top, q_i)$.

First we construct a set of trap languages such that a configuration without any agents with state $q_{\bot}$ marking all the corresponding traps has to encode a correct tiling with Wang tiles. For technical reasons, we will encode the tiling without using the agent with the index 0, which will have state $q_1$. We consider the indices $1, 2, \ldots, n - 1$ to all currently execute the loop transition $t$. We consider the relationship that agent $i$ inspects agent $j$ next in $i$-th execution of $t$ to be a mapping of $i$ to $j$. We have now access to two relations between agents: their natural order and the mapping induced by $t$. We use these two relations to construct a grid of the following form:
As a set of configurations specified by traps has to be closed under transitions, we also include all configurations where at least one agent has the state $q_\perp$. We now proceed to define the necessary trap languages and prove they ensure the desired structure.

For convenience we will omit “or some agent has state $q_\perp$” when defining the properties of global configurations. We will also omit the slots where all letters have $0$. All the trap languages we present consist of normalized traps, because the transitions can either change $\text{badflag}$ or move some agent into the state $q_\perp$; the former never changes whether the traps are marked, and the latter ensures that all the traps we present are marked.

First we ensure that the agents have correct locations. This is ensured by the following trap languages:

$$\left( \langle \square \rangle \{ q_l, q_\perp \} \right)^* \left( \langle \square \rangle \{ q_\perp \} \right)^* \text{index} \quad \text{loc}$$

$$\left( \langle \square \rangle \{ q_l \} \right)^* \left( \langle \square \rangle \{ q_\perp \} \right)^* \left( \langle p_1 \rangle \{ q_l \} \right)^* \left( \langle \square \rangle \{ q_\perp \} \right)^* \text{index} \quad \text{loc} \quad t.p_1$$

It is easy to see that these trap languages require that the first agent is in the state $q_l$ while the remaining agents are iterating and not examining the agent with the index 0, not themselves. (Or some agent is in the state $q_\perp$)

For convenience, let us also exclude the case when the two relations coincide or are reverse of each other. This corresponds to the following trap languages.

$$\left( \langle \square \rangle \{ q_\perp \} \right)^* \left( \langle \square \rangle \{ q_l \} \right)^* \left( \langle p_1 \rangle \{ q_\perp \} \right)^* \left( \langle \square \rangle \{ q_l \} \right)^* \text{index} \quad \text{loc} \quad t.p_1$$

$$\left( \langle \square \rangle \{ q_\perp \} \right)^* \left( \langle \square \rangle \{ q_l \} \right)^* \left( \langle \square \rangle \{ q_\perp \} \right)^* \left( \langle p_1 \rangle \{ q_l \} \right)^* \left( \langle \square \rangle \{ q_\perp \} \right)^* \text{index} \quad \text{loc} \quad t.p_1$$

$$\left( \langle \square \rangle \{ q_\perp \} \right)^* \left( \langle \square \rangle \{ q_l \} \right)^* \left( \langle \square \rangle \{ q_\perp \} \right)^* \left( \langle p_1 \rangle \{ q_l \} \right)^* \text{index} \quad \text{loc} \quad t.p_1$$
Next we ensure that, roughly speaking, $t(k + 1) = t(k) + 1$, with wraparound from $n$ to 1. The general case is handled by the following two trap languages.

\[
\begin{pmatrix}
\{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset
\end{pmatrix}
\]

\[
\begin{pmatrix}
\{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset
\end{pmatrix}
\]

\[
\begin{pmatrix}
\{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset
\end{pmatrix}
\]

\[
\begin{pmatrix}
\{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset
\end{pmatrix}
\]

\[
\begin{pmatrix}
\{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset
\end{pmatrix}
\]

We can interpret this language as follows. Given the agents with the indices $i$, $i + 1$, $j$, $j + 1$, either the agent with the index $i$ is not inspecting the agent with the index $j$, or the agent with the index $i + 1$ is inspecting the agent with the index $j + 1$. In other words, if the agent number $i$ inspects the agent number $j$, then the agent number $i + 1$ inspects the agent number $j + 1$ (assuming $i + 1$ and $j + 1$ do not exceed $n - 1$).

The two special cases correspond to $i$ or $j$ being equal to $n - 1$. These are described by the following two trap languages.

\[
\begin{pmatrix}
\{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset
\end{pmatrix}
\]

\[
\begin{pmatrix}
\{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset
\end{pmatrix}
\]

\[
\begin{pmatrix}
\{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset
\end{pmatrix}
\]

\[
\begin{pmatrix}
\{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} & \{q_1\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset
\end{pmatrix}
\]

Given that no agent inspects itself or its neighbour, the trap languages defined by now ensure that inspection relation is just addition of a constant modulo $n - 1$.

We now proceed to define the trap languages that ensure that the edges of the grid are correctly marked. First we consider the first column. Informally speaking, the first colour contains the agents reachable from the first one by steps corresponding to the inspection relation. First let us ensure that all such agents have the $firstCol$ variable set. The following language ensures that the agent with the index 1 has this variable set.

\[
\begin{pmatrix}
\{q_1\} & \{q_1\} & \{q_1\} \\
\emptyset & \{q_1\} & \emptyset
\end{pmatrix}
\]

The following two languages ensure correct interactions with the inspection relation.
What such traps ensure is: for any distinct indices $i$ and $j$, either the $i$-th agent does not have $firstCol$ equal to $\top$, or it is not currently inspecting the $j$-th agent, or $j$-th agent has $firstCol$ equal to $\top$. In other words, any agent inspected by an agent with $firstCol$ set also has $firstCol$ set. We now need to ensure that only the agents in the first column have the $firstCol$ variable equal to $\top$. To do this, we demand that no agent between the agent with the index 1 and the agent inspected by the agent with the index 1 has $firstCol$ set. This is ensured by the following trap language.

Indeed, using the conversion from disjunction to implication we can say that any agent between 1 and $t(1)$ has $firstCol$ unset. In particular, this implies that $t(1) - 1$ divides $n - 1$ (otherwise the wraparound would create a violation). The last column can now be defined as the last agent plus all the agents immediately before the agents in the first column using the following trap languages.

The latter two languages say that an agent in the middle has $lastCol$ if and only if the next agent has $firstCol$ set. The first row and the last row can now be defined as all the agents between the first agent inclusive and its inspectee non-inclusive, and between the last agent’s inspector non-inclusive and the last agent inclusive. This corresponds to the following four trap languages.
assert the compatibility of the placed Wang-tiles. For this we write $\{\tau\}$ to denote the set $\mathbb{T} \setminus \{\tau\}$ and $E(\tau) = \{\tau' \in \mathbb{T} \mid \tau(E) = \tau'(W)\}$; i.e., $E(\tau)$ is the set of tiles that can be placed east of $\tau$ (similarly for $W(\tau), N(\tau)$ and $S(\tau)$). In the following, consider for the following definitions of languages $\tau$ a parameter and the languages unions for all $\tau \in \mathbb{T}$. We start with horizontal compatibility. For two adjacent indices $i$ and $i + 1$ where agent $i$ does not claim to be on the last column this is straightforward:

These languages and observations suffice to enforce a grid in the desired structure. It remains to assert the compatibility of the placed Wang-tiles. For this we write $\{\tau\}$ to denote the set $\mathbb{T} \setminus \{\tau\}$ and $E(\tau) = \{\tau' \in \mathbb{T} \mid \tau(E) = \tau'(W)\}$; i.e., $E(\tau)$ is the set of tiles that can be placed east of $\tau$ (similarly for $W(\tau), N(\tau)$ and $S(\tau)$). In the following, consider for the following definitions of languages $\tau$ a parameter and the languages unions for all $\tau \in \mathbb{T}$. We start with horizontal compatibility. For two adjacent indices $i$ and $i + 1$ where agent $i$ does not claim to be on the last column this is straightforward:

Indeed, what such traps ensure is that whenever the agent with the index $i$ is not in the last column and it has tile $\tau$, the next agent has one of the tiles from $E(\tau)$. We only check compatibility from left to right, which is sufficient as long as we check each pair of adjacent tiles at least in one direction.

For the agents from the last column, though, the next agent to the right is actually next agent after the corresponding inspector, or the inspector of the agent with the next index. These two definitions coincide when both are applicable, but in the first row and in the last row only one of the two works. We thus use the following trap languages.
Thus these traps ensure that the safety condition $\psi$ has one agent in the state $q_{\perp}$ use it to build a configuration where no agent is in the state $q_{\perp}$. Now the trap languages provide for each $p_{1}$ $\psi_{\text{badflag}}$ has one agent in the state $q_{\perp}$ use it to build a configuration where no agent is in the state $q_{\perp}$.

The question whether some trap languages ensure safety, or at least inductiveness of safety, is at least as hard as the existence of a periodic tiling using given Wang tiles, and thus undecidable.
C \textit{k}-rendezvousing systems

On closer inspection of the proof of Theorem 1 one can see that the argument mainly relies on the possibility to identify an agent $m$ which is not involved in the transition and remove it. Pointer variables might cause transitions to interact with more agents; namely, those agents pointer variables currently point to. However, since there are only finitely many pointer variables, the amount of agents a transition might interact with is still finite. In the following we expand the definitions of $\text{move}^T_{m}[-c]$ and $\text{move}^T_{m}[-c]$ to allow pointer slots in the set $T$. Then, we formalize this in the notion of $k$-rendezvousing systems. Essentially, this means that any transition only relies on the state of $k$ agents. Formally, we say

\textbf{Definition 9.} We call any parameterized system $\mathcal{S}$ with loop transitions $T_{lp}$ and pointer variables $\text{Ptrs}$ \textit{k-rendezvousing} if for every instance of any transition $c \vdash c'$ there are $k$ indices $\{i_1, \ldots, i_k\}$ such that

- $c_\ell = c'_\ell$ for all $\ell \in [|c|] \setminus \{i_1, \ldots, i_k\}$, and
- $(\text{drop}_m \circ \text{move}_m^{T_{lp} \cup \text{Ptrs}[\ell]})(c) \vdash \text{move}_m^{T_{lp} \cup \text{Ptrs}[\ell]}(c')$ is an instance of a transition for every $m \in [|c|] \setminus \{i_1, \ldots, i_k\}$, and
- $(\text{drop}_m \circ \text{move}_m^{T_{lp} \cup \text{Ptrs}[\ell]})(c) \vdash \text{move}_m^{T_{lp} \cup \text{Ptrs}[\ell]}(c')$ is an instance of a transition for every $m \in [|c|] \setminus \{i_1, \ldots, i_k\}$.

With this definition we can give a straightforward generalization of Theorem 1:

\textbf{Theorem 3.} Let $\mathcal{S}$ be a $k$-rendezvousing parameterized system. Let $\mathcal{N}_{0} \ldots \mathcal{N}_{n-1} \in \Sigma_{\mathcal{S}}$ be a normalized trap of the instance of $\mathcal{S}$ with $n$ agents. If $\mathcal{N}_i(\text{index}) = \Box$ and $\mathcal{N}_i = \mathcal{N}_{i+1} = \ldots = \mathcal{N}_{i+k-1}$, then for every $\ell \geq 1$ the word

$\mathcal{N}_{0} \ldots \mathcal{N}_{i-1} \mathcal{N}_{i}^{\ell_0} \mathcal{N}_{i+1} \ldots \mathcal{N}_{n-1}$

is a normalized trap of the instance with $n + \ell - 1$ agents.

\textbf{Proof.} For the sake of contradiction we assume the statement of the theorem to be incorrect. Then we can fix a minimal $\ell_0 \geq 1$ for which it does not hold true anymore. Let $O$ be the instance of $\mathcal{N}_{0} \ldots \mathcal{N}_{i-1} \mathcal{N}^{\ell_0} \mathcal{N}_{i+1} \ldots \mathcal{N}_{n-1}$. If $\ell_0 = 1$ then this already contradicts with the assumption of the theorem. Thus, consider the case that $\ell_0 > 1$. Then, however, we have $O_i = O_{i+1} = \ldots = O_{i+\ell_0+k}$. Consequently, there exists $m \in \{i, \ldots, i + \ell_0 + k\} \setminus \{i_1, \ldots, i_k\}$. Consider the instance $O'$ of $\mathcal{N}_{0} \ldots \mathcal{N}_{i-1} \mathcal{N}_{i}^{(\ell_0 - 1)} \mathcal{N}_{i+1} \ldots \mathcal{N}_{n-1}$ which – by minimality of $\ell_0$ – is indeed a trap and $\text{drop}_m(O) = O'$. By the choice of $m$ there is an adjacent index $m'$ of $m$ such that $m' \in \{i, \ldots, i + \ell_0 + k\}$. W.l.o.g. $m' = m - 1$ (otherwise exchange $\text{move}_m^{T_{lp} \cup \text{Ptrs}[\ell]}$ with $\text{move}_m^{T_{lp} \cup \text{Ptrs}[\ell]}$ in the following). Since $O_m = O_{m'}$ we have $c \cap O$ if and only if $\text{move}_m^{T_{lp} \cup \text{Ptrs}[\ell]}(c) \cap O$ and $c' \cap O$ if and only if $\text{move}_m^{T_{lp} \cup \text{Ptrs}[\ell]}(c') \cap O$. Since $\mathcal{S}$ is $k$-rendezvousing and by choice of $m$ we know $(\text{drop}_m \circ \text{move}_m^{T_{lp} \cup \text{Ptrs}[\ell]})(c) \vdash (\text{drop}_m \circ \text{move}_m^{T_{lp} \cup \text{Ptrs}[\ell]})(c')$. Since $O$ is not a trap we know $c \cap O$ while $c' \cap O$. By our observation above this means $\text{move}_m^{T_{lp} \cup \text{Ptrs}[\ell]}(c) \cap O$ but $\text{move}_m^{T_{lp} \cup \text{Ptrs}[\ell]}(c') \cap O$. This implies, since $c_m = c'_m$ that $c_m(s) \notin O_m(s)$ for all slots $s$. Now, we can see that $O'$ cannot be a trap since $\text{drop}_m(O) = O'$ and, thus, $(\text{drop}_m \circ \text{move}_m^{T_{lp} \cup \text{Ptrs}[\ell]})(c) \cap O'$ but $(\text{drop}_m \circ \text{move}_m^{T_{lp} \cup \text{Ptrs}[\ell]})(c') \cap O'$. This renders $\ell_0$ not minimal in contradiction to the assumption. The statement of the theorem follows. \qed