Invariant Generation for Parametrized Systems using Self-Reflection

Alejandro Sanchez, Sriram Sankaranarayanan, César Sánchez, and Bor-Yuh Evan Chang

University of Colorado Boulder
Department of Computer Science
Technical Report

CU-CS-1094-12

June 2012

Department of Computer Science
430 UCB
Boulder, Colorado 80309-0430

www.cs.colorado.edu

Invariant Generation for Parametrized Systems using Self-Reflection

(Extended Version)

Alejandro Sanchez[1], Sriram Sankaranarayanan[2], César Sánchez[1,3], and Bor-Yuh Evan Chang[2]. ⋆

[1] IMDEA Software Institute, Madrid, Spain
{first.lastname}@imdea.org
[2] University of Colorado, Boulder, CO, USA
{first.lastname}@colorado.edu
[3] Institute for Applied Physics, CSIC, Spain

Abstract. We examine the problem of inferring invariants for parametrized systems. Parametrized systems are concurrent systems consisting of an *a priori* unbounded number of process instances running the same program. Such systems are commonly encountered in many situations including device drivers, distributed systems, and robotic swarms. In this paper we describe a technique that enables leveraging off-the-shelf invariant generators designed for sequential programs to infer invariants of parametrized systems. The central challenge in invariant inference for parametrized systems is that naïvely exploding the transition system with all interleavings is not just impractical but impossible. In our approach, the key enabler is the notion of a *reflective abstraction* that we prove has an important correspondence with inductive invariants. This correspondence naturally gives rise to an iterative invariant generation procedure that alternates between computing candidate invariants and creating reflective abstractions.

1 Introduction

We study the problem of automatically inferring invariants for parametrized systems. Parametrized systems are multi-threaded programs that may be executed by a finite but unbounded number of thread instances executing in parallel. The individual thread instances belonging to the same process type execute the same set of program instructions involving local variables that are unique to each thread instance, as well as the global shared variables. Parametrized programs are useful in many settings including device drivers, distributed algorithms, concurrent data structures, robotic swarms, and biological systems. The thread instances in a parametrized program communicate through shared memory and synchronization mechanisms including locks, synchronous rendezvous, and broadcast communication.

⋆ This work was supported in part by the US National Science Foundation (NSF) under grants CNS-0953941 and CCF-1055066; the EU project FET IST-231620 *HATS*, MICINN project TIN-2008-05624 *DOVES*, CAM project S2009TIC-1465 *PROMETIDOS*, and by the COST Action IC0901 *Rich ModelToolkit-An Infrastructure for Reliable Computer Systems*.

This manuscript is an extended version of a paper appearing in SAS 2012. The additional material in this version are fully contained in the appendix and include proofs and details about our empirical evaluation.

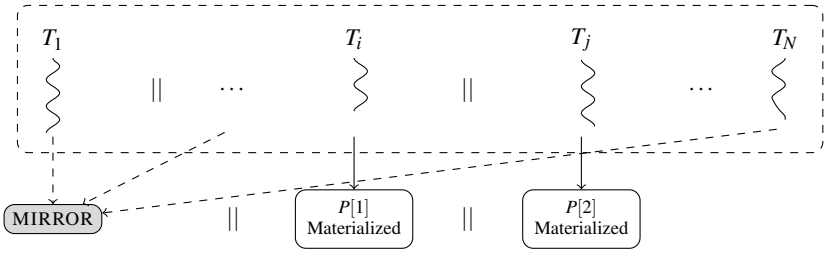

Fig. 1. A reflective abstraction to infer 2-indexed invariants of a parametrized system. We abstract the system as two materialized processes and the mirror process.

In this paper, we define an abstract-interpretation–based framework for inferring indexed invariants of parametrized programs. A k-indexed invariant of a parametrized system is an invariant over the local variables of an arbitrary k distinct thread instances and the global shared variables. The main idea is to build what we call a *reflective abstraction* of the parametrized program that consists of a fixed number of *materialized* processes composed with a *mirror* abstraction that summarizes the effect of the remaining thread instances on the global variables. In Fig. 1, we hint at this construction for deriving 2-indexed invariants (and discussed further in Sect. 2).

We show how invariants computed at various program locations of the materialized processes can be transferred in a suitable way into guards of the mirror process. In this way, the abstraction of other interfering threads via the mirror process varies during the course of the analysis — much like how materialization in shape analysis enables the heap abstraction to vary for better precision. Our approach can be viewed as an abstract interpretation over the cartesian product of the abstract domain of state assertions over program variables and the reflective abstractions of the environment. This allows us to cast existing methods for invariant generation for parametrized systems [4,30] as different iteration schemes for computing fixed points. Finally, we define new iteration schemes and compare their effectiveness empirically. In summary, we arrive at a characterization of invariants of the parametrized program as fixed points of a monotone functional that (a) computes a (post-) fixed point on the reflective abstraction and (b) transfers the fixed point to the guards of the mirror processes.

Overall, this paper contains the following contributions. We present the notion of a *reflective abstraction*, which gives a means to summarize the effects of other threads dynamically during the analysis (Sect. 3). We then formally prove a correspondence between reflective abstractions and inductive invariants, which leads naturally to an iterative invariant generation procedure that allows leveraging off-the-shelf invariant generators for sequential programs (Sect. 4.1). We discuss how the reflective abstraction framework encompasses interference abstractions (Sect. 4.2), as well as practical considerations for an implementation (Sect. 4.3). Finally, we present some initial experience on evaluating applications of our reflective abstraction framework (Sect. 5). In particular, we study three variants of reflective abstraction and one variant of interference abstraction and how they compare with respect to the invariants they obtain. We

```
   global data: int array(len) where len > 0, next: int where next = 0;
   thread P {
     local c: int where c = 0, end: int where end = 0;
0    atomic if (next + 10 <= len) { c := next; next := end := next + 10; }
1    while (c < end) {
2       assert(0 <= c && c < len); data[c]:= ... process data[c] ...;
3       c := c + 1;
4    }
   }
```

Fig. 2. WORKSTEAL: A parametrized array processing program. Each thread processes a "chunk" with 10 elements.

find that surprisingly, widening appears to have a less predictable effect for parametrized systems than for sequential systems.

2 Overview: Self-Reflection

In this section, we illustrate the basic idea behind reflective abstractions of parametrized systems, and we give a sense of how such a construction enables inference of k-indexed invariants. Consider the program WORKSTEAL in Fig. 2. Parametrized programs consist of a fixed but unbounded number of thread instances $T_1, \ldots, T_N$ where $N \geq 1$. In the rest of this paper, we use $[N]$ to denote the set of indices $\{1, \ldots, N\}$. Each thread runs the set of statements in P. In this program, there is a global array data of size len and a global variable next that holds the current unprocessed index. Each instance T_i has thread local variables $\mathsf{c}[i]$ and $\mathsf{end}[i]$ for $i \in [N]$, that is, local variables are replicated for and indexed by each thread instance. The local variable $\mathsf{c}[i]$ holds a current index of the data element being processed by the thread instance T_i. The variable $\mathsf{end}[i]$ holds the limiting index for thread T_i.

Our goal is to prove properties about the behavior of parametrized systems that must hold regardless of the number of running thread instances N. The simplest properties involve only global variables, such as $\psi_0 \colon (\mathsf{next} \mod 10 = 0)$. Other properties may also involve local variables, as well as globals. In case local variables are involved, we differentiate instances of local variables by indexing them. A 1-indexed property refers to a local variable from a single thread instance. In our example, we wish to prove the 1-indexed property corresponding to the assertion at location 2:

$$\psi_1 \colon (\forall\, i)\ 0 \leq \mathsf{c}[i] < \mathsf{len} \qquad \text{(i.e., access of array data is in bounds).} \tag{1}$$

An example of a 2-indexed property is where we wish to establish race-freedom for *distinct* thread instances i_1, i_2 whenever one of the instances resides at location 2:

$$\psi_2 \colon (\forall\, i_1, i_2)\ \ \mathsf{c}[i_1] \neq \mathsf{c}[i_2] \qquad \text{(i.e., access of array data is race free).} \tag{2}$$

We will use $i, i_1, i_2, \ldots$ to refer to process instances ranging within the set of thread indices $[N]$. We will assume implicitly that different symbols i_j, i_k involved in a given

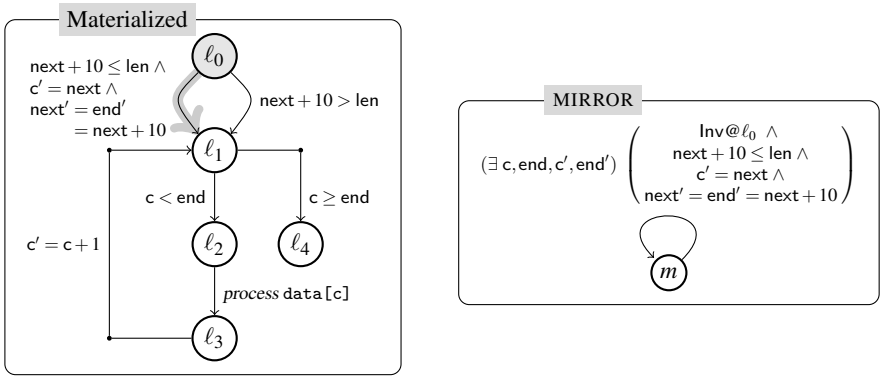

Fig. 3. Transition system models for a materialized thread and the MIRROR process. The guard $\mathsf{Inv}@\ell_0$ for the MIRROR transition comes from the invariant computed at location ℓ_0 in a materialized thread.

assertion ψ are used to refer to different process instances (e.g., there is an implicit pre-condition that $i_1 \neq i_2$ in ψ_2).

In this paper, we adapt existing invariant synthesis techniques to parametrized programs. Our technique allows us to generate invariants such as ψ_0, ψ_1, and ψ_2 for the parametrized program in Fig. 2. Our approach, inspired by the idea of materialization in shape analysis [33], is based on identifying a fixed number of *materialized processes* and summarizing the remaining processes into a single, separate process that we will call the MIRROR. We show this idea pictorially in Fig. 1 where a parametrized system with N thread instances is modeled by three threads: 2 materialized thread instances $P[1], P[2]$ and the mirror process that summarizes the effects of the $N-2$ remaining thread instances on the shared global variables. The number of materialized processes is fixed *a priori* based on the desired form of the invariants. For example, we need to materialize at least 2 threads to infer 2-indexed invariants. The novel aspect of our *reflective* approach is that the MIRROR process is *not* fixed *a priori* but rather is derived as part of the fixed point analysis.

Fig. 3 shows the basic setup for invariant synthesis for the WORKSTEAL program. The composition of the materialized thread(s) and the MIRROR yields a regular sequential transition system that can be analyzed using a standard abstract interpretation engine. The MIRROR process simulates the effect that the remaining (non-materialized) threads in the system have on the shared variables next and len. In particular, the MIRROR process has no local variables. The running example has a single transition from location ℓ_0 to ℓ_1 that affects the shared variable next (highlighted), and variable len is never updated anywhere. This transition is copied as a self-loop around a single location in the MIRROR process, and the local variable updates are quantified away. However, to maintain precision, it is preferable to restrict the scope of this transition only to those states of the program that are actually reachable at location ℓ_0. We will over-approximate these states by an assertion $\mathsf{Inv}@\ell_0$. The main question is then to precisely determine what $\mathsf{Inv}@\ell_0$ is. A simple solution is to assume $\mathsf{Inv}@\ell_0$: *true* to

yield a valid over-approximation of all states that are reachable whenever some process resides at location ℓ_0, but *true* is very often a coarse over-approximation. Our key observation is that $\mathsf{Inv}@\ell_0$ and correspondingly the construction of the MIRROR process need not be fixed *a priori*. Instead, we build a more precise abstraction by incrementally constructing MIRROR as follows. The first iteration sets $\mathsf{Inv}@\ell_0 : \mathit{false}$, in effect, disabling the mirror. This iteration approximates only those states reachable by the materialized threads running in isolation. Subsequently, we run an abstract interpreter and compute invariants of the composition of the current MIRROR process and the materialized threads. The MIRROR process for the next iteration is updated with $\mathsf{Inv}@\ell_0$ set to the candidate invariants computed at location ℓ_0 in the materialized threads with the local variables projected out. This *candidate invariant reflection* allows the MIRROR to run from a larger portion of the reachable state space. Convergence is achieved whenever the invariants obtained at some iteration are subsumed by those at the previous iteration. At this point, the effect of the mirror and the materialized processes in the invariants and the guards is stable. Upon convergence, we obtain k-indexed invariants that relate the local variables of the k materialized threads to the global variables.

3 Reflective Abstractions and Inductive Invariants

In this section, we define the notion of a *reflective abstraction* of a parametrized system. We first present the basic model of parametrized systems. The main result of this section (Theorem 1) proves the soundness of reflective abstractions.

A parametrized system consists of a large, *a priori* unbounded set of processes that (a) run the same sequence of instructions and (b) interact with each other through some synchronization primitives. The model presented here is based on concurrent systems communicating through shared memory. For convenience, we use the fair transition system model [27]. To simplify the presentation further, all program variables are assumed to be of integer type.

A *parametrized transition system* Π is described by $\langle G, X, \mathit{Trs}, \ell_0, \Theta \rangle$ consisting of a set of shared (global) variables $g \in G$, a set of local variables $x \in X$, a finite set of locations $\ell \in \mathit{Loc}$, a finite set of transitions $\tau \in \mathit{Trs}$, an initial location ℓ_0, and an initial condition Θ that denotes the set of possible initial values of the global and local variables. A transition $\tau : \langle \ell_{\text{src}}, \ell_{\text{tgt}}, \rho \rangle$ consists of a pre-location ℓ_{src}, a post-location ℓ_{tgt} and a transition relation ρ that relates the values of the variables (global and local) before the transition with the values after it. We use primed variables (e.g., $g' \in G'$ and $x' \in X'$) to refer to the values of the corresponding variables in the post-location.

Example 1 (Parametrized Transition System). Consider the WORKSTEAL program from Fig. 2. Its corresponding parametrized transition system Π consists of globals $G = \{\mathsf{len}, \mathsf{next}\}$, locals $X = \{\mathsf{c}, \mathsf{end}\}$, and locations $\mathit{Loc} = \{\ell_0, \ell_1, \ell_2, \ell_3, \ell_4\}$ with the initial location being ℓ_0. Ignoring the MIRROR process, Fig. 3 depicts the transition relations *Trs* of the parametrized transition system (solid edges). Not modeling the array `data` means the transition relation ρ between ℓ_2 and ℓ_3 is a no-op, that is, $\rho = \text{preserve}(G \cup X)$. We define $\text{preserve}(Z) \stackrel{\text{def}}{=} \wedge_{z \in Z}\, z' = z$ for any set of variables Z, that is, the transition relation where all variables in Z are preserved.

The semantics of a parametrized system is given with respect to a positive number N of *thread instances*. The overall *state* of a parametrized system with N thread instances is described by valuations of the shared (global) variables, the *local variable instances*, and the *location instances* of each thread. That is, a local variable instance $x[i]$ is the instance of local variable $x \in X$ for thread instance $i \in [N]$. The set $X[i]$ refers to the local variable instances of thread instance i.

A state $\sigma\colon \langle L, V\rangle$ is characterized by a map $L : [N] \rightharpoonup_{\text{fin}} Loc$ that associates a location $L(i)$ for each thread instance i and a valuation map V that maps each shared (global) variable and each local variable instance to its integer value. We write $V(G)$ to denote the valuations to all global variables and $V(X[i])$ to denote the valuations of locals of thread i. We write $V \models \varphi$ for a valuation V satisfying a formula φ. Where helpful for clarity in presentation, we write $\varphi[G,X]$ to indicate the variables over which the formula φ is defined. A *run* of a parametrized system instantiated with N thread instances is a finite or infinite sequence of states such that (1) the initial state satisfies the initial condition, and (2) a step between two successive states is obtained by executing one transition in one thread instance. More detailed definitions of parametrized systems and their runs are given in Appendix A (Definition 3 and Definition 4, respectively).

Definition 1 (1-Indexed Invariant). *A pair $\langle \ell, \varphi\rangle$ consisting of a location ℓ and assertion $\varphi[G,X]$ is a 1-index invariant of a parametrized program Π iff for every reachable state $\sigma\colon (L,V)$ with $N > 0$ thread instances, and for every $i \in [N]$*

$$\text{if } L(i) = \ell \text{ then } (V(G), V(X[i])) \models \varphi\,.$$

In other words, the valuations of the local variables $X[i]$ and global variables G for any thread instance i reaching the location ℓ satisfies φ.

The notion of 1-indexed invariants generalizes to k-indexed invariants involving the global variables and the local variables of some $k > 0$ threads. This generalization is given explicitly in Appendix A (Definition 5).

Example 2 (k-Indexed Invariants). Property ψ_1 (see (1) on page 3) is an example of a 1-indexed invariant for the parametrized system in Example 1 (i.e., $\langle \ell_2, 0 \leq \mathsf{c} < \mathsf{len}\rangle$). Property ψ_2 (see (2) in page 3) corresponds to many 2-indexed invariants at different pairs of locations. Each invariant is of the form $\langle \ell_2, \_, \mathsf{c}[1] \neq \mathsf{c}[2]\rangle$ or $\langle \_, \ell_2, \mathsf{c}[1] \neq \mathsf{c}[2]\rangle$ where $\_$ refers to any location. These invariants say that one of the thread instances resides at location ℓ_2 (and the other anywhere else) with the respective instances of the local variable c holding different values.

Reflective Abstractions. In essence, a reflective abstraction is an over-approximation of a parametrized transition system by a sequential one. What makes an abstraction *reflective* is that the over-approximation is computed *under an assertion map*.

We skip the formal definition of sequential transition systems, noting that these correspond to single thread instances of parametrized transition systems. To denote specifically a sequentially transition system, we use the meta-variable Σ. Let $\Gamma[X]$ be some fixed first-order language of assertions, such as the theory of integer arithmetic, involving free variables X. We overload $\models$ to denote the semantic entailment relation

between these formulae. An *assertion map* $\eta : Loc \rightharpoonup_{\text{fin}} \Gamma[X]$ maps each location to an assertion in $\Gamma[X]$. An assertion map η is *inductive* whenever (1) the assertion at the initial location subsumes the initial condition (i.e., initiation) and (2) the assertion map respects the (strongest) post-condition transformer (i.e., consecution): for any transition τ between ℓ_{src} and ℓ_{tgt}, the post-condition transformer for τ applied to $\eta(\ell_{\text{src}})$ entails $\eta(\ell_{\text{tgt}})$. Standard definitions for the post-condition transformer and inductive assertion maps are given in Appendix A (Definition 6 and Definition 7, respectively). Inductive invariants are fundamental to the process of verifying safety properties of programs. In order to prove an assertion φ over all reachable states at a location ℓ, we seek an inductive assertion map η over the entire program such that $\eta(\ell) \models \varphi$. The map η is termed an *inductive strengthening* of φ.

We now formally define the notion of a *reflective abstraction*, which abstracts a parametrized system by a system with a $k > 0$ materialized processes, and a MIRROR process that models the "interference" of the remaining threads on the shared variables G. Our key result is that *an invariant of a reflective abstraction is that of a parametrized system*. To simplify the presentation, the rest of this section will describe reflective abstractions with a single materialized thread (i.e., $k = 1$). Our definitions readily extend to the case when $k > 1$.

Let η be an assertion map over the locations of a parametrized system Π. Our goal is to define a sequential system $\text{REFLECT}_{\Pi}(\eta)$. Such a system will contain transitions to model one specific thread instance, termed the *materialized thread*, and the MIRROR process, which models the influence of the other threads on the shared variables.

Definition 2 (Reflective Abstraction). *The* reflective abstraction *of a parametrized system* Π: $\langle G, X, Loc, Trs, \ell_0, \Theta \rangle$ *with respect to an assertion map* η *is a sequential transition system, written* $\text{REFLECT}_{\Pi}(\eta)$*, over variables* $G \cup X$*, with locations given by* Loc *and transitions given by* $Trs \cup \{ \text{MIRROR}(\tau, \eta, \ell) \mid \tau \in Trs \text{ and } \ell \in Loc \}$.
The original transitions Trs *model the materialized thread, while the* MIRROR *transitions model the visible effects of the remaining threads.*

For transition τ: $\langle \ell_{\text{src}}, \ell_{\text{tgt}}, \rho \rangle$ *and some location* $\ell \in Loc$*, the corresponding* MIRROR *transition* $\text{MIRROR}(\tau, \eta, \ell)$ *is defined as follows:*

$$\left\langle \ell,\ \ell,\ \text{preserve}(X) \wedge (\exists\, Y, Y')\ \left(\underline{\eta(\ell_{\text{src}})[G,Y]} \wedge \rho[G,Y,G',Y']\right) \right\rangle.$$

Finally, the initial location of the reflective abstraction is ℓ_0 *and the initial condition* Θ *(i.e., comes directly from the parametrized system).*

Note that each MIRROR transition is a self-loop at location ℓ of the materialized thread, or equivalently, MIRROR can be seen as a process with a single location and self-looping transitions that is composed with the materialized thread. Note that each MIRROR transition preserves the local variables of the materialized thread. Also, observe that the (underlined) guard of the MIRROR transition includes the invariant $\eta(\ell_{\text{src}})$ of the interfering thread at the pre-location, which can be seen as reflecting the invariant of the materialized thread at ℓ_{src} on to the interfering thread. Finally, the local variables are projected away from the transition relation using existential quantification to model the effect of the materialized transition on the shared variables.

Example 3 (Reflective Abstraction). The following table shows a part of an assertion map η for the program in Fig. 2, along with the corresponding mirror transitions computed from it, that is, of $\text{REFLECT}_{\Pi}(\eta)$. We write $\rho(\tau)$ for the transition relation of transition τ (in the original parametrized system Π) and $\rho(m)$ for the transition relation of a MIRROR transition in the reflective abstraction. Note that the assertion map η is not necessarily inductive.

Name	Invariant/Relation
$\eta(\ell_0)$	$\mathsf{next} = 0 \ \wedge\ \mathsf{c} = 0 \ \wedge\ \mathsf{end} = 10$
$\rho(\tau_0\colon \langle \ell_0, \ell_1, \rho_0 \rangle)$	$\mathsf{next} + 10 \leq \mathsf{len} \ \wedge\ \mathsf{c}' = \mathsf{next} \ \wedge\ \mathsf{next}' = \mathsf{end}' = \mathsf{next} + 10$ $\wedge\ \text{preserve}(\{\mathsf{len}\})$
$\rho(m_0\colon \text{MIRROR}(\tau_0, \eta, \_))$	$\mathsf{next} = 0 \ \wedge\ 10 \leq \mathsf{len} \ \wedge\ \mathsf{next}' = 10 \ \wedge\ \text{preserve}(\{\mathsf{len}, \mathsf{c}, \mathsf{end}\})$
$\rho(\tau_0'\colon \langle \ell_0, \ell_1, \rho_0' \rangle)$	$\mathsf{next} + 10 > \mathsf{len} \ \wedge\ \text{preserve}(\{\mathsf{next}, \mathsf{len}, \mathsf{c}, \mathsf{end}\})$
$\rho(m_0'\colon \text{MIRROR}(\tau_0', \eta, \_))$	$10 > \mathsf{len} \ \wedge\ \text{preserve}(\{\mathsf{next}, \mathsf{len}, \mathsf{c}, \mathsf{end}\})$
$\eta(\ell_3)$	$\mathsf{next} \geq 0 \ \wedge\ \mathsf{c} \geq 0 \ \wedge\ \mathsf{c} < \mathsf{end}$
$\rho(\tau_3\colon \langle \ell_3, \ell_1, \rho_3 \rangle)$	$\mathsf{c}' = \mathsf{c} + 1 \ \wedge\ \text{preserve}(\{\mathsf{next}, \mathsf{len}, \mathsf{end}\})$
$\rho(m_3\colon \text{MIRROR}(\tau_3, \eta, \_))$	$\mathsf{next} \geq 0 \ \wedge\ \text{preserve}(\{\mathsf{next}, \mathsf{len}, \mathsf{c}, \mathsf{end}\})$
$\eta(\ell_4)$	$\mathsf{next} \geq 0 \ \wedge\ \mathsf{c} \geq 10$

The transition relation of the MIRROR transition m_0 is derived from the original transition τ_0 by computing: $\text{preserve}(\{\mathsf{c}, \mathsf{end}\}) \wedge ((\exists\, \mathsf{c}, \mathsf{end}, \mathsf{c}', \mathsf{end}')\ \eta(\ell_0) \wedge \rho_0)$. Eliminating the existential quantifier from

$$\text{preserve}(\{\mathsf{c}, \mathsf{end}\}) \wedge (\exists\, \mathsf{c}, \mathsf{end}, \mathsf{c}', \mathsf{end}') \left[\begin{array}{c} \mathsf{next} = 0 \ \wedge\ c = 0 \ \wedge\ \mathsf{end} = 10 \ \wedge \\ \mathsf{next} + 10 \leq \mathsf{len} \ \wedge\ \ c' = \mathsf{next} \ \wedge\ \ \mathsf{next}' = \mathsf{end}' = \mathsf{next} + 10 \ \wedge \\ \text{preserve}(\{\mathsf{len}\}) \end{array} \right]$$

yields the MIRROR transition relation of m_0 shown above. We note that other MIRROR transitions preserve the global variables next and len (e.g., m_0' or m_3). Thus, these transitions may be omitted from the MIRROR process while preserving all behaviors. Mirror transition m_0 is the one illustrated in Fig. 3.

We now present the main result involving reflective abstractions: if η is an inductive invariant of the reflective abstraction $\text{REFLECT}_{\Pi}(\eta)$ for a parametrized program Π, then for every location ℓ, the assertion $\eta(\ell)$ is a 1-indexed invariant (Cf. Definition 1).

Theorem 1 (Reflection Soundness). *Let η be an assertion map such that η is inductive for the system $\text{REFLECT}_{\Pi}(\eta)$. It follows that for each location ℓ of Π, $\eta(\ell)$ is a* 1*-index invariant.*

The proof proceeds by induction on the runs of the parametrized system Π. The full proof is provided in Appendix A. To summarize, if one discovers a map η that is inductive for the system $\text{REFLECT}_{\Pi}(\eta)$, then we may conclude that $\eta(\ell)$ is a 1-index invariant for location ℓ in Π. In spite of its circularity, this characterization naturally suggests that the process of constructing a suitable η can be cast as a fixed point and solved using abstract interpretation.

To generalize reflective abstraction to $k > 1$ materialized threads, we first construct a transition system that is the product of k-copies of the parametrized program Π. This transition system uses k-copies of the locals and a single instance of the globals from Π. Then, given an assertion map η, we add MIRROR transitions to construct the reflective abstraction following Definition 2 on this product system. Each transition τ is projected onto the (global) shared variables guarded by the assertion given by η in the transition's pre-location. An inductive assertion derived on the reflective abstraction of the product system is a k-indexed invariant for the original parametrized system.

4 Reflective Abstract Interpretation

In this section, we present an iterative procedure to generate invariants of a parametrized system by applying abstract interpretation on reflective abstractions. We explain a *lazy* and an *eager* approach to reflective abstract interpretation and contrast reflective abstraction with *interference abstraction*, a commonly-used approach when analyzing multi-thread programs (e.g., [30]).

First, we briefly recall the theory of abstract interpretation [13,14,5] for finding inductive assertion maps as the fixed point of a monotone operator over an *abstract domain*. Abstract interpretation is based on the observation that invariants of a program are over-approximations of the *concrete collecting semantics* η^*, an assertion map that associates each location ℓ with a first-order assertion $\eta^*(\ell)$ characterizing all reachable states at the location ℓ. Formally, we write $\eta^* = \mathsf{lfp}\ \mathscr{F}_\Sigma(\mathit{false})$. Here, $\mathscr{F}_\Sigma(\eta)$ is a "single-step" semantics—a monotone operator over the lattice of assertion maps that collects all the states reachable in at most one step of the system Σ, and *false* maps every location to *false*. For this presentation, we will rewrite slightly that familiar equation, making the transition system Σ an explicit argument of $\mathscr{F}$ (rather than fixed):

$$\eta^* = \mathsf{lfp}\ \mathscr{F}(\mathit{false}, \Sigma)\,. \tag{3}$$

We can also define a *structural pre-order* on sequential transition systems. We say Σ structurally refines Σ', written $\Sigma \preceq \Sigma'$, as simply saying that Σ and Σ' have the same structure—in terms of their variables, locations, and transitions—and where the initial conditions and the corresponding transition relations are ordered by $\models$. It is clear that if $\Sigma \preceq \Sigma'$, then the behaviors of Σ' over-approximate the behaviors of Σ. A more detailed definition is given in Appendix B. Now, we can see that the concrete collecting semantics functional $\mathscr{F}(\eta, \Sigma)$ is monotone over both arguments: (a) over concrete assertion maps ordered by $\models$ location-wise and (b) over sequential transition systems using the structural pre-order.

The abstract interpretation framework allows one to approximate the *collecting semantics* of programs in an abstract domain $\mathscr{A}$: $\langle A, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ defined by a lattice. The abstract lattice is related to the concrete lattice of first-order assertions $\Gamma[X]$ through a *Galois connection* described by an abstraction function $\alpha : \Gamma[X] \to A$ that maps assertions in the concrete domain to abstract objects and $\gamma : A \to \Gamma[X]$ that interprets abstract objects as concrete assertions representing sets of states. In the abstract interpretation framework, we lift the operator $\mathscr{F}$ defined over the concrete domain to the corresponding monotone operator $\widehat{\mathscr{F}}$ over the abstract domain $\mathscr{A}$. Analogously, we

write $\widehat{\eta} : \mathit{Loc} \rightharpoonup_{\text{fin}} A$ for an abstract assertion map. A fixed point computation in (3) is then expressed in terms of the abstract domain $\mathscr{A}$ as follows: $\widehat{\eta}^* = \mathsf{lfp}\ \widehat{\mathscr{F}}(\bot, \Sigma)$. Here, $\bot$ is the abstract assertion map that maps every location to the bottom element of the abstract domain $\bot$. If $\mathscr{A}$ is an abstract domain, then it follows that $\gamma \circ \widehat{\eta}^*$ yields an inductive assertion map over the concrete domain.

If the domain $\mathscr{A}$ is finite or has the *ascending chain condition*, the least-fixed point lfp operator may be computed iteratively. On the other hand, many domains of interest fail to satisfy these conditions. Herein, abstract interpretation provides us a framework using the *widening* operator that can be repeatedly applied to guarantee convergence to a *post-fixed point* that over-approximates the least-fixed point. Concretizing this post-fixed point leads to a valid (but weaker) inductive assertion map.

4.1 Abstract Interpretation using Reflection

The overall idea behind our invariant generation technique is to alternate between constructing a (sequential) reflective abstraction of the given parametrized system Π and applying abstract interpretation for sequential systems on the reflective abstraction. We distinguish two abstract interpretation schemes: *lazy* and *eager*.

Lazy Reflective Abstract Interpretation. Lazy reflective abstract interpretation for a parametrized system Π proceeds as follows: First, begin with an initial abstract candidate invariant map $\widehat{\eta}_0$ that maps each location to the least abstract element $\bot$. Then, iterate the following steps until convergence: (a) compute the reflective abstraction Σ_j using $\widehat{\eta}_j$; (b) on the reflective abstraction Σ_j, apply an abstract interpreter for sequential systems to obtain the next candidate invariant map $\widehat{\eta}_{j+1}$; (c) terminate the iteration whenever $\widehat{\eta}_{j+1}(\ell) \sqsubseteq \widehat{\eta}_j(\ell)$ for all $\ell \in \mathit{Loc}$. We now proceed formally to derive the lazy abstract interpretation scheme above. Let $\widehat{\mathscr{G}}_{\text{LAZY},\Pi}$ be the following operator defined over the abstract lattice:

$$\widehat{\mathscr{G}}_{\text{LAZY},\Pi}(\widehat{\eta}) \stackrel{\text{def}}{=} \mathsf{lfp}\ \widehat{\mathscr{F}}(\bot, \textsc{Reflect}_{\Pi}(\gamma \circ \widehat{\eta})). \tag{4}$$

Given a map $\widehat{\eta}$ associating locations with abstract objects, the operator $\widehat{\mathscr{G}}_{\text{LAZY},\Pi}$ is implemented by (a) concretizing $\widehat{\eta}$ to compute $\textsc{Reflect}_{\Pi}(\gamma \circ \widehat{\eta})$, the reflective abstraction; and (b) applying the least fixed point of $\widehat{\mathscr{F}}$ over the reflection. We note that the monotonicity of $\widehat{\mathscr{G}}_{\text{LAZY}}$ holds where lfp is computable. In particular, we note that $\textsc{Reflect}_{\Pi}(\eta)$ is a monotone operator. The overall scheme for inferring invariants of the original system Π consists of computing the following:

$$\widehat{\eta}^* = \mathsf{lfp}\ \widehat{\mathscr{G}}_{\text{LAZY},\Pi}(\bot) \qquad \text{and let map } \eta_{\text{inv}} \stackrel{\text{def}}{=} \gamma \circ \widehat{\eta}^*. \tag{5}$$

Soundness follows from the soundness of abstract interpretation and reflection soundness (Theorem 1). In practice, we implement the operator $\widehat{\mathscr{G}}_{\text{LAZY}}$ by constructing a reflective abstraction and calling an abstract interpreter as a black-box. Note that if the abstract interpreter uses widening to enforce convergence, $\widehat{\mathscr{G}}_{\text{LAZY}}$ is not necessarily monotone since the post-fixed point computation cannot be guaranteed to be monotone. We revisit these considerations in Sect. 4.3.

Eager Reflective Abstract Interpretation. In contrast with the lazy scheme, it is possible to construct an eager scheme that weaves the computation of a least-fixed point

and the reflective abstractions in a single iteration. This scheme can be thought of as abstract interpretation on the Cartesian product of the abstract domain $\mathscr{A}$ and the space of reflective abstractions $\textsc{Reflect}_{\Pi}(\gamma \circ \widehat{\eta})$ for $\widehat{\eta} \in (Loc \rightharpoonup_{\text{fin}} A)$ ordered by the structural pre-order relation $\preceq$.

The eager scheme consists of using an eager operator and an eager reflective abstract interpretation as a least-fixed point computation with that operation starting at $\bot$:

$$\widehat{\mathscr{G}}_{\text{EAGER},\Pi}(\widehat{\eta}) \stackrel{\text{def}}{=} \widehat{\mathscr{F}}(\widehat{\eta}, \textsc{Reflect}_{\Pi}(\gamma \circ \widehat{\eta})) \qquad \text{and} \qquad \widehat{\eta}^* = \mathsf{lfp}\, \widehat{\mathscr{G}}_{\text{EAGER},\Pi}(\bot). \quad (6)$$

In other words, we apply a single step of the abstract operator $\widehat{\mathscr{F}}$ starting from the map $\widehat{\eta}$ over the reflective abstraction from $\gamma \circ \widehat{\eta}$.

4.2 Interference Abstraction versus Reflective Abstraction

We compare and contrast the eager and lazy reflective abstraction approaches with the commonly used interference abstraction. The goal of interference abstraction (see for example [30]) is to capture the effect of interfering transitions flow-insensitively much like a reflective abstraction. The *interference semantics* can be expressed concisely in the formalism developed in this section by the following operator:

$$\widehat{\eta}^* = \mathsf{lfp}\, \widehat{\mathscr{F}}(\bot, \Sigma_\top) \quad \text{where } \Sigma_\top \stackrel{\text{def}}{=} \textsc{Reflect}_{\Pi}(\mathit{true}). \quad (7)$$

Here $\top$ represents the abstract assertion map that associates each location with $\top \in A$. In particular, the mirror process is fixed to say that any transition in Π (i.e., of an interfering thread) can fire at any point.

As a concrete example, consider the parametrized system with a global variable g and a local variable x shown on the right. At location 0, a thread waits until the value of the global g is positive and then saves that value into its local variable x while setting g to 0. It then increments that value saved locally and writes it back to the global g signaling completion of its processing. Our first goal is to establish that $g \geq 0$ everywhere.

```
   global g: int where g >= 0;
   thread P {
     local x: int where x = 0;
0    atomic { await(g > 0);
       x := g; g := 0; }
1    x := x + 1;
2    atomic { g := x; }
3  }
```

Consider the transition from location ℓ_2 to ℓ_3. Following the framework described in this paper, the ideal transition relation for the corresponding MIRROR transition is $((\exists\, x)\ \eta^*(\ell_2) \wedge g' = x)$. The interference semantics over-approximates $\eta^*(\ell_2)$ with *true*, so this interference transition is simply a non-deterministic update to g, which causes a failure to derive $g \geq 0$ anywhere. In contrast, the reflective abstraction approach described in this paper over-approximates η^* incrementally starting from $\bot$ in the abstract domain. Doing so enables inferring invariants on x that can then be used to derive $g \geq 0$—in particular, using that $x > 0$ for any thread instance at location ℓ_2. However, the reflective abstraction approach is not complete either. For instance, reflective abstractions cannot be used to establish the invariant $g = 0$ when all threads are at location ℓ_1 or ℓ_2 without the use of additional auxiliary variables.

4.3 Theory versus Practice: The Effect of Widening

Thus far in this section, we have defined all iterations via least-fixed points of monotone operators, implicitly assuming abstract domains for which the least-fixed point is computable. However, in practice, abstract interpretation is used with abstract domains that do not enjoy this property. In particular, we want to be able to use abstract domains that rely on widening to enforce convergence to a post-fixed point that over-approximates the least-fixed point.

Applying abstract interpretation with widening instead of lfp in the previous definitions of this section raises a number of issues in an implementation. First, the lazy reflective operator $\widehat{\mathscr{G}}_{\text{LAZY}}$ defined in (4) on page 10 is not necessarily monotonic. To remedy this in our implementation we enforce monotonicity by applying an "outer join" that joins the assertion maps from the previous iteration with the one from the current iteration. To enforce convergence of this iteration, we must apply an "outer widening" should the abstract domain warrant it.

Another consequence is that the relative precision of the reflective abstract interpretation schemes are unclear. Perhaps counter-intuitively, the interference abstraction approach described in Sect. 4.2 is not necessarily less precise than the reflective abstract interpretation with $\widehat{\mathscr{G}}_{\text{EAGER}}$ as defined in (6). To see this possibility, let $\widehat{\eta}^*_{\text{EAGER}}$ be the fixed point abstract assertion map computed by iterating $\widehat{\mathscr{G}}_{\text{EAGER}}$. While the final reflective abstraction $\Sigma_{\text{EAGER}} : \text{REFLECT}_{\Pi}(\gamma \circ \widehat{\eta}^*_{\text{EAGER}})$ using $\widehat{\mathscr{G}}_{\text{EAGER}}$ is trivially no less precise than the interference abstraction $\Sigma_{\text{INTERFERE}} : \text{REFLECT}_{\Pi}(\mathit{true})$, the abstract interpretation with widening is not guaranteed to be monotonic. Instead, this observation suggests another scheme, which we call eager+. The eager+ scheme runs $\widehat{\mathscr{G}}_{\text{EAGER}}$ to completion to get Σ_{EAGER} and then applies standard abstract interpretation over this sequential transition system. In other words, the eager+ scheme is defined as follows:

$$\widehat{\eta}^*_{\text{EAGER}} = \mathsf{lfp}\, \widehat{\mathscr{G}}_{\text{EAGER},\Pi}(\bot) \qquad \widehat{\eta}^*_{\text{EAGER+}} = \mathsf{lfp}\, \widehat{\mathscr{F}}(\bot, \text{REFLECT}_{\Pi}(\gamma \circ \widehat{\eta}^*_{\text{EAGER}})) \,. \quad (8)$$

5 Empirical Evaluation: Studying Iteration Schemes

We present here an empirical evaluation of implementations of the eager, eager+, lazy, and interference schemes. The main questions that we seek to answer are: (a) how effective are each of these schemes at generating invariants of interest, and (b) how do the invariants generated by each scheme compare with each other in terms of precision? We also look at performance of the analyses secondarily.

Methodology. We consider a set of five benchmarks, including a simple barrier algorithm [29], a centralized barrier [29], the work stealing algorithm presented in Fig. 2, a generalized version of dinning philosophers with a bounded number of resources, and a parametrized system model of autonomous swarming robots inside a $m \times n$ grid [12]. They range in size from 2–75 locations, 6–24 variables, and 4–49 transitions. For each problem, we specify a set of target invariants, with the intention to check whether the automatically generated invariants imply a given program's safety specification. The number of target invariants ranges from 4–16. Our study focuses on examining the technique space rather than the benchmark space, so we do not discuss the details of the benchmarks. Those details are available in Appendix C.

Table 1. Timing and precision results for Lazy, Eager, Eager+ and Interference abstract interpretations. Legend: **ID:** benchmark identifier, **Dom:** abstract domains, **I:** intervals, **O:** octagons, **P:** polyhedra, **Prps**: total number of properties to be proven, **Time**: seconds, **Prp**: number of properties proved, **TO**: timed out ($\geq$ 1.5 hours), **Wid**: number of widening iterations (*) for the lazy scheme we report external widening applications.

ID	Dom	Prps	Lazy			Eager			Eager+			Interf.		
			Time	Wid*	Prp	Time	Wid	Prp	Time	Wid	Prp	Time	Wid	Prp
Tbar	I	4	0.1	2	0	0.1	5	0	0.1	5	0	0.1	4	0
	P		0.2	4	4	0.1	5	4	0.1	5	4	0.1	4	4
	O		0.8	3	3	0.1	5	3	0.1	5	3	0.1	4	3
Wsteal	I	5	0.3	6	2	0.1	5	1	0.1	5	1	0.1	4	0
	P		2.4	6	1	0.1	7	1	0.2	7	3	0.1	7	5
	O		8.2	6	4	7.5	6	4	0.2	6	4	6.2	5	4
Cbar	I	9	0.9	3	4	0.1	7	0	0.1	8	0	0.1	7	0
	P		TO		0	1.7	11	4	2.7	12	5	1.1	10	6
	O		TO		0	7.5	9	6	11.3	9	6	6.2	8	4
Phil	I	14	1.9	4	2	0.1	8	2	0.1	8	2	0.1	7	0
	P		11.8	6	14	1.1	11	8	1.8	11	8	6.3	13	14
	O		TO		0	25	12	4	40	12	4	20	12	4
Rb(2,2)	I	16	31.3	8	4	0.4	10	4	0.4	11	4	0.2	10	0
	P		TO		0	9.3	22	3	15	23	3	5.8	15	4
	O		TO		0	142	25	3	225	26	3	105	18	3
Rb(2,3)	I	18	133	8	6	0.7	10	6	0.9	11	6	0.5	10	0
	P		TO		0	23	22	5	36.8	23	5	16	15	5
	O		TO		0	404	25	5	629	26	5	320	18	5
Rb(3,3)	I	23	1141	8	9	1.6	10	9	2.1	11	9	0.9	10	0
	P		TO		0	68.2	22	8	111.5	23	8	52	15	8
	O		TO		0	1414	25	8	2139	26	8	1168	18	8
Rb(4,4)	I	29	TO		0	6.7	11	16	9.4	11	16	3.2	11	0
	P		TO		0	49	23	15	396	23	15	303	15	15
	O		TO		0	TO		0	TO		0	TO		0

We have implemented the reflective and interference abstraction schemes in the LEAP theorem proving framework for verifying functional correctness properties of parametrized programs, currently being developed at the IMDEA Software Institute. The approaches proposed here extend LEAP by generating invariant assertions automatically. After compiling a parametrized program written in an imperative language into a transition system, we generate inductive assertions using the lazy, eager, and eager+ reflective abstraction schemes and the interference abstraction scheme. Our framework directly uses the abstract domains implemented in the Apron library [23]. Narrowing is used for the eager, eager+, and interference schemes but not the lazy scheme.

Results. Table 1 presents a comparison of timings and precision across the lazy, eager, eager+, and interference schemes. The running time in seconds is given for each method under the Time columns. While interference abstractions are the fastest, as expected, it is perhaps surprising to note that the lazy scheme was markedly slower than the remaining techniques considered. In fact, it times out on many instances. Likewise, we note that eager and eager+ were only slower by a factor of 1.1–1.5 on most benchmarks when compared to interference abstraction. Also surprisingly, the time for using polyhedra is generally faster than octagons. According to the Apron authors, the execution time of

polyhedra can vary widely between good and bad cases, while the worst case and best case execution time of octagons is the same, which may explain this observation.

The properties proved by each method are given under the Prp columns. Again, surprisingly, the interference semantics fares noticeably better than the other schemes for the polyhedral domain but noticeably worse on the interval domain. Also, the interval domain itself seems to fare surprisingly better than the polyhedral domain in terms of properties proved. In many cases, however, the properties proved by these domains were non-overlapping. Perhaps the simplest explanation for this result is that the properties themselves mostly concern proving bounds on variables. It is not surprising that the interval domain can establish this. Yet another factor is the use of polyhedral widening. Since a widening needs to be carried out at every location in the program, the loss of precision in the polyhedral domain can be considerable.

In Table 2, we compare each pair of methods in terms of the relative strengths of the invariants inferred. Some surprising patterns are revealed. For one, lazy (L), eager (E), and eager+ (E+) prove stronger invariants for the interval domain when compared to the interference (In) scheme. On the other hand, the trend is reversed for the polyhedral domain. In many cases, the invariants are either incomparable or invariants of one technique are stronger at some location and weaker at others. Conjoining the invariants in these cases can produce stronger invariants overall.

Table 2. Comparing the strength of the inference. For a comparison A:B, $+$ means A's invariants are stronger than B in at least one location and not weaker elsewhere (conversely for $-$), $=$ means the same everywhere, and $\neq$ means incomparable somewhere.

ID	Dom	L:E	L:E+	L:In	E:In	E+:In	E:E+
Tbar	I	$-$	$-$	$+$	$+$	$+$	$=$
	P	$=$	$=$	$+$	$+$	$+$	$=$
	O	$=$	$=$	$+$	$+$	$+$	$=$
Wsteal	I	$+$	$+$	$+$	$+$	$+$	$=$
	P	$+$	$\neq$	$\neq$	$\neq$	$\neq$	$-$
	O	$=$	$=$	$+$	$+$	$+$	$=$
Cbar	I	$\neq$	$\neq$	$+$	$+$	$+$	$=$
	P	TO	TO	TO	$\neq$	$\neq$	$-$
	O	TO	TO	TO	$+$	$+$	$=$
Phil	I	$+$	$+$	$+$	$+$	$+$	$=$
	P	$+$	$+$	$+$	$-$	$-$	$=$
	O	TO	TO	TO	$+$	$+$	$=$
Rb(2,2)	I	$+$	$+$	$+$	$+$	$+$	$=$
	P	TO	TO	TO	$\neq$	$\neq$	$-$
	O	TO	TO	TO	$\neq$	$+$	$-$
Rb(2,3)	I	$+$	$+$	$+$	$+$	$+$	$=$
	P	TO	TO	TO	$\neq$	$\neq$	$-$
	O	TO	TO	TO	$\neq$	$+$	$-$
Rb(3,3)	I	$+$	$+$	$+$	$+$	$+$	$=$
	P	TO	TO	TO	$\neq$	$\neq$	$-$
	O	TO	TO	TO	$\neq$	$+$	$-$
Rb(4,4)	I	TO	TO	TO	$+$	$+$	$=$
	P	TO	TO	TO	$\neq$	$\neq$	$-$
	O	TO	TO	TO	TO	TO	TO

Interpretation of Results. In theory, all the methods presented can be viewed as post-fixed point computations in the product domain representing sets of states and reflective abstractions. Our intuition with abstract interpretation suggests that the interference scheme, which applies a single iteration on the sequential system generated from the $\top$ reflection, should fare worse than the eager scheme which computes a least fixed point using Kleene iteration. The comparison results are quite surprising, however. We conclude that widening and the associated non-monotonicity play a significant role for parametrized systems. This effect is much more so than for sequential systems, wherein, our past experience suggests that non-monotonicity of widening plays a more limited role. A future direction of research might focus on minimizing the use of widenings or avoiding them altogether using constraint-based techniques [11] or recent advances based on policy and strategy iterations [19,20].

6 Related Work

The problem of verifying parametrized systems has received a lot of attention in recent years. This problem is, in general, undecidable [3]. However, numerous decidable subclasses have been identified [7,15,21,16,25,8,1,2,6]. Our approach here is an instance of the general framework of thread-modular reasoning [18,22,26,9], wherein one reasons about a thread in isolation given some assumptions about its environment (i.e., the other concurrently executing threads). Notably, the approach considered here builds up the assumptions incrementally via self-reflection.

One of the main issues in verifying parametrized programs is the interaction between a given thread and its environment, consisting of the remaining threads. Abstracting this interaction finitely has been considered by many, recently by Berdine et al. [4] and Farzan et al. [17]. In particular, the approach of Berdine et al. is very closely related. Similarities include the notion of transferring invariants from a materialized thread to the abstraction of the remaining threads. However, Berdine et al. do not explicitly specify an iteration scheme, that is, how the inferred candidate invariants are transferred to the environment abstraction. Furthermore, the effects of widening, including the potential non-monotonicity in many domains, are not studied. As observed in this paper, such considerations have a significant impact on the generated invariants. Another recent contribution is that of Farzan et al. that explores the interleaving of control and data-flow analyses to better model the thread interference in parametrized programs. In our framework, their setup roughly corresponds to the lazy scheme. However, Farzan et al. do not incrementally consider the transference of data properties, and instead they focus on ruling out infeasible interferences due to control.

The idea of abstracting away the effects of interacting threads by projecting away the local variables is quite standard. The recent work of Miné et al. [30] analyzes multi-threaded embedded systems using this abstraction. Likewise, Kahlon et al. present a framework for the abstract interpretation of multi-threaded programs with finitely-many threads. Therein, a *melding operator* is used to model the effect of an interfering thread on the abstract state of the current thread [24].

Our approach presented here does not explicitly handle synchronization constructs such as locks and pairwise rendezvous. These constructs can be handled using the framework of *transaction delineation* presented by Kahlon et al. [24]. Here, a single-threaded sequential analysis pass is first carried out to identify sections of the program which can be executed "atomically" while safely ignoring the interferences by the remaining threads. Exploring the use of the delineated transactions to construct the reflective abstraction in the framework of this paper is a promising future direction that will enable us to analyze larger and more complex software systems.

Another class of approaches relies on finite model properties wherein invariants of finite instantiations generalize to the parametrized system as a whole. One such approach is that of *invisible invariants* pioneered by Pnueli et al. [32,35]. This approach finds inductive invariants by fixing the number of processes and computing invariants of the instantiated system. These invariants are heuristically generalized to the parametrized system, which are then checked to be inductive. In [28], invisible invariants are generalized in the abstract interpretation framework as fixed points. In specific instances, a finite model property is used to justify the completeness of this technique. A

related method is that of *splitting invariants* [31,10] that ease the automation of invariant generation but also assumes finite state processes and the existence of a cut-off [15].

7 Conclusion

We have described the *reflective abstraction* approach for inferring k-indexed invariants of parametrized systems. This approach was inspired partly by the notions of materialization-summarization from shape analysis. The central idea was that inferences made on materialized threads can be transferred or *reflected* on to the summarized threads (i.e., the MIRROR process). This perspective not only suggests a new technique but describes a space of possible invariant inference techniques, including previously-defined interference abstractions. As such, we studied three variants of reflective abstraction that we defined and the interference abstraction to better understand their relative strength in inferring invariants. To our surprise, our study revealed what appears to be a significant amount of unpredictability in invariant inference strength as the result of widening. The effect of widening seems to be larger for reflective abstract interpretation of parametrized systems than for standard abstract interpretation of sequential systems. We hypothesize that the presence of loops at each program location (from the composition with the MIRROR process) is the primary culprit behind this observation, suggesting a direction for future inquiry. Another future direction is to examine how additional structure can be imposed on the summarized threads.

References

1. Abdulla, P.A., Bouajjani, A., Jonsson, B., Nilsson, M.: Handling global conditions in parametrized system verification. In: CAV'99. LNCS, vol. 1633, pp. 134–145. Springer (1999)
2. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized verification of infinite-state processes with global conditions. In: CAV'07. LNCS, vol. 4590, pp. 145–157. Springer (2007)
3. Apt, K.R., Kozen, D.C.: Limits for automatic verification of finite-state concurrent systems. Info. Proc. Letters 22(6), 307–309 (1986)
4. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, S.: Thread quantification for concurrent shape analysis. In: CAV'08. LNCS, vol. 5123, pp. 399–413. Springer (2008)
5. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software (invited chapter). In: The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones. LNCS, vol. 2566, pp. 85–108. Springer (2005)
6. Bozzano, M., Delzanno, G.: Beyond parameterized verification. In: TACAS'02. LNCS, vol. 2280, pp. 221–235. Springer (2002)
7. Clarke, E.M., Grumberg, O., Browne, M.C.: Reasoning about networks with many identical finite-state processes. In: PODC'86. pp. 240–248. ACM (1986)
8. Clarke, E.M., Grumberg, O., Jha, S.: Veryfying parameterized networks using abstraction and regular languages. In: CONCUR'95. LNCS, vol. 962, pp. 395–407. Springer (1995)
9. Clarke, E.M., Talupur, M., Veith, H.: Proving Ptolemy right: The environment abstraction framework for model checking concurrent systems. In: TACAS'08. LNCS, vol. 4963, pp. 33–47. Springer (2008)

10. Cohen, A., Namjoshi, K.S.: Local proofs for global safety properties. FMSD 34(2), 104–125 (2009)
11. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: CAV'03. LNCS, vol. 2725, pp. 420–433. Springer (2003)
12. Correll, N., Martinoli, A.: Collective inspection of regular structures using a swarm of miniature robots. In: ISER. Springer Tracts in Advanced Robotics, vol. 21, pp. 375–386. Springer (2004)
13. Cousot, P., Cousot, R.: Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL'77. pp. 238–252. ACM (1977)
14. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to Abstract interpretation, invited paper. In: PLILP'92. LNCS, vol. 631, pp. 269–295. Springer (1992)
15. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: CADE'00. LNAI, vol. 1831, pp. 236–254. Springer (2000)
16. Emerson, E.A., Namjoshi, K.S.: Automatic verification of parameterized synchronous systems. In: CAV'96. LNCS, vol. 1102, pp. 87–98. Springer (1996)
17. Farzan, A., Kincaid, Z.: Verification of parameterized concurrent programs by modular reasoning about data and control. In: POPL'12. pp. 297–308. ACM (2012)
18. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: SPIN'03. LNCS, vol. 2648, pp. 213–224. Springer (2003)
19. Gaubert, S., Goubault, E., Taly, A., Zennou, S.: Static analysis by policy iteration on relational domains. In: ESOP'07. LNCS, vol. 4421, pp. 237–252. Springer (2007)
20. Gawlitza, T., Seidl, H.: Precise fixpoint computation through strategy iteration. In: ESOP'07. LNCS, vol. 4421, pp. 300–315. Springer (2007)
21. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. of the ACM 39(3), 675–735 (1992)
22. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread-modular abstraction refinement. In: CAV'03. LNCS, vol. 2725, pp. 262–274. Springer (2003)
23. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV'09. LNCS, vol. 5643, pp. 661–667. Springer (2009)
24. Kahlon, V., Sankaranarayanan, S., Gupta, A.: Semantic reduction of thread interleavings in concurrent programs. In: TACAS. Lecture Notes in Computer Science, vol. 5505, pp. 124–138. Springer (2009)
25. Lesens, D., Halbwachs, N., Raymond, P.: Automatic verification of parameterized linear networks of processes. In: POPL'97. pp. 346–357. ACM (1997)
26. Malkis, A., Podelski, A., Rybalchenko, A.: Precise thread-modular verification. In: SAS'07. LNCS, vol. 4634, pp. 218–232. Springer (2007)
27. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer (1995)
28. McMillan, K.L., Zuck, L.D.: Invisible invariants and abstract interpretation. In: SAS'11. LNCS, vol. 6887, pp. 249–262. Springer (2011)
29. Mellor-Crummey, J.M., Scott, M.L.: Barriers for the sequent symmetry, `ftp://ftp.cs.rochester.edu/pub/packages/scalable_synch/locks_and_barriers/Symmetry.tar.Z`
30. Miné, A.: Static analysis of run-time errors in embedded critical parallel C programs. In: ESOP'11. LNCS, vol. 6602, pp. 398–418 (2011)
31. Namjoshi, K.S.: Symmetry and completeness in the analysis of parameterized systems. In: VMCAI'07. LNCS, vol. 4349, pp. 299–313. Springer (2007)
32. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: TACAS'01. LNCS, vol. 2031, pp. 82–97. Springer (2001)

33. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. ACM Trans. Program. Lang. Syst. 20(1), 1–50 (1998)
34. Sánchez, C.: Deadlock Avoidance for Distributed Real-time and Embedded Systems. Ph.D. thesis, Computer Science Department, Stanford University (May 2007)
35. Zuck, L.D., Pnueli, A.: Model checking and abstraction to the aid of parameterized systems (a survey). Computer Languages, Systems & Structures 30, 139–169 (2004)

A Soundness of Reflective Abstractions

Definition 3 (Parametrized Transition Systems). *A* parametrized transition system Π *with shared variables G and local variables X is given by a tuple $\langle G, X, Loc, Trs, \ell_0, \Theta\rangle$, wherein*

1. *Set $Loc = \{\ell_0, \ldots, \ell_m\}$ denotes a finite set of locations.*
2. *Set $Trs = \{\tau_1, \ldots, \tau_k\}$ denotes a finite set of transitions. Each transition τ is a tuple $\langle \ell_{\text{src}}, \ell_{\text{tgt}}, \rho\rangle$ with the following components:*
   (a) *location ℓ_{src} is the pre-location of the transition, which we also write as* $\text{src}(\tau)$*— the source of transition τ;*
   (b) *location ℓ_{tgt} is the post-location of the transition, which we also write as* $\text{tgt}(\tau)$*— the target of transition τ; and*
   (c) *relation ρ is a* transition relation *that relates the values of the global and local variables (G, X) before the transition to the values (G', X') after the transition; we write $\rho[G, X, G', X']$ to indicate the variables involved.*
3. *Location $\ell_0 \in Loc$ denotes the initial location.*
4. *Assertion $\Theta[G, X]$, called the* initial condition *denotes the set of possible initial values of the global and local variables.*

Definition 4 (Runs of Parametrized Systems). *A* run *of the instantiated parametrized system is defined as a finite or infinite sequence of states*

$$\sigma_0 \xrightarrow{\tau_1[i_1]} \sigma_1 \xrightarrow{\tau_2[i_2]} \sigma_2 \cdots \xrightarrow{\tau_m[i_m]} \sigma_m \cdots \qquad \textit{such that}$$

1. *The initial state $\sigma_0 \colon \langle L_0, V_0\rangle$ satisfies the initial condition. That is,*

$$L_0(i) = \ell_0 \quad \textit{and} \quad V_0(G), V_0(X[i]) \models \Theta \qquad \textit{for all } i \in [N].$$

2. *The step $\sigma_j \colon \langle L_j, V_j\rangle \xrightarrow{\tau_j[i_j]} \sigma_{j+1} \colon \langle L_{j+1}, V_{j+1}\rangle$ between a state and its successor obtained by executing transition τ_j on thread instance i_j satisfies that:*
   (a) *The locations of thread i_j in σ_j and σ_{j+1} match the pre- and post-locations of τ_j, respectively (i.e., $L_j(i_j) = \text{src}(\tau_j)$ and $L_{j+1}(i_j) = \text{tgt}(\tau_j)$).*
   (b) *The valuations of the global variables and local variables of thread instance i_j are related by the transition relation. That is,*

$$V_j(G), V_j(X[i_j]), V_{j+1}(G), V_{j+1}(X[i_j]) \models \rho_{\tau_j}[G, X, G', X'].$$

   (c) *For all other thread instances the locations and local variables remain unchanged between σ_j and σ_{j+1}:*

$$L_j(i) = L_{j+1}(i) \;\; \textit{and} \;\; V_j(x[i]) = V_{j+1}(x[i]) \qquad \textit{for all } i \neq i_j, \textit{ and } x \in X.$$

Definition 5 (k-Indexed Invariant). *A tuple $\langle \ell_1, \ell_2, \ldots, \ell_k, \varphi\rangle$ consisting of locations $\ell_1, \ldots, \ell_k$ and assertion $\varphi[G, X_1, \ldots, X_k]$, where $X_1, \ldots, X_k$ are k disjoint copies of the local variables, is a k-indexed invariant of a parametrized transition system Π whenever for every reachable state $\sigma \colon (L, V)$ with $N \geq k$ thread instances, and for every $i_1, \ldots, i_k \in [N]$ where $i_a \neq i_b$ if $a \neq b$, if $L(i_1) = \ell_1$, $L(i_2) = \ell_2$, $\cdots$, and $L(i_k) = \ell_k$ then*

$$(V(G), V(X[i_1]), \cdots, V(X[i_k])) \models \varphi(G, X_1, \ldots, X_k).$$

Definition 6 (Post-Condition). *Let φ be an assertion over X and τ be a transition with transition relation ρ_τ. The (strongest) post-condition (also known as image, transfer function, or transformer) of φ across τ is given by the assertion*

$$\mathrm{post}(\varphi,\tau)\colon\ (\exists\, X_0)\ (\ \varphi[X_0]\ \wedge\ \rho_\tau[X_0,X]\ )\,.$$

The post-condition describes all the valuations of variables X reachable in one step by executing transition τ starting from a state that satisfies φ.

Definition 7 (Inductive Assertion Map). *An* assertion map $\eta : Loc \rightharpoonup_{fin} \Gamma[X]$ *maps each location ℓ to an assertion. An assertion map η is* inductive *whenever the following conditions hold:*

− **Initiation:** *The assertion at ℓ_0 subsumes the initial condition:*

$$\Theta\ \models\ \eta(\ell_0)\,.$$

− **Consecution:** *For each transition $\tau\colon\ \langle \ell_{\mathrm{src}}, \ell_{\mathrm{tgt}}, \rho\rangle$,*

$$\mathrm{post}(\eta(\ell_{\mathrm{src}}),\tau)\ \models\ \eta(\ell_{\mathrm{tgt}})\,.$$

Theorem 1 (Reflection Soundness). *Let η be an assertion map such that η is inductive for the system* $\textsc{Reflect}_\Pi(\eta)$. *It follows that for each location ℓ of Π, $\eta(\ell)$ is a* 1*-index invariant.*

Proof. We will prove this property by induction over natural number the m, considering all the states reachable in m steps.

For the base case, we consider all states reachable in 0 steps. Let $N > 0$ represent the number of thread instances. Each initial state is of the form L_0, V_0 where $L_0(i) = \ell_0$ and $V_0(G), V_0(X[i]) \models \Theta(G,X)$. Since, $\Theta \models \eta(\ell_0)$, we have $L_0(i) = \ell_0$ and $V_0(G), V_0(X[i]) \models \eta(\ell_0)$. This verifies the base case.

Let us assume that for every state $\sigma_m : (L_m, V_m)$ reachable in some $m \geq 0$ steps by $N > 0$ thread instances,

$$V_m(G), V_m(X[i]) \models \eta(L_m(i)), \quad \text{for all } i \in [N]\,.$$

Consider an arbitrary state $\sigma_{m+1} : (L_{m+1}, V_{m+1})$ reachable in one step from σ_m by the execution of some transition $\tau[i_m]$ for $i_m \in [N]$, i.e., $\sigma_m \xrightarrow{\tau[i_m]} \sigma_{m+1}$. We wish to prove that

$$V_{m+1}(G), V_{m+1}(X[j])\ \models\ \eta(L_{m+1}(j)), \quad \text{for all } j \in [N]\,.$$

The proof splits into two cases:

(a) $j = i_m$. By our induction hypothesis we have $V_m(G), V_m(X[i_m]) \models \eta(L_m(i_m))$ and furthermore,

$$V_m(G), V_m(X[i_m]), V_{m+1}(G), V_{m+1}(X[i_m])\ \models\ \rho_\tau\,.$$

Therefore, $V_{m+1}(G), V_{m+1}(X[i_m]) \models\ \mathrm{post}(\eta(L_m(i_m)),\tau)$. Applying consecution of η for τ, we get

$$V_{m+1}(G), V_{m+1}(X[i_m]) \models\ \mathrm{post}(\eta(L_m(i_m)),\tau)\ \models\ \eta(L_{m+1}(i_m))\,.$$

(b) $j \neq i_m$. Let $j \neq i_m$ refer to some thread instance whose local state remains unchanged due to the execution of $\tau[i_m]$:

$$V_m(X[j]) = V_{m+1}(X[j])\,.$$

The execution of $\tau[i_m]$ can be treated as a transition of the MIRROR process. In particular, the local variables of the thread instance i_m simulated by the MIRROR process satisfy $\eta(L_m(i_m))$ by the induction hypothesis. Therefore, we have

$$(V_m(G), V_{m+1}(G)) \models (\exists\, X, X')\ \eta(L_m(i_m)) \ \wedge\ \rho_\tau(G, X, G', X')\,.$$

Therefore,

$$\begin{aligned} V_m(G), V_m(X[j]), V_{m+1}(G), V_{m+1}(X[j]) \ \models \\ \underbrace{\text{preserve}(X) \ \wedge\ (\exists\, Y, Y')\ \eta(L_m(i_m)) \ \wedge\ \rho_\tau(G, Y, G', Y')}_{\text{MIRROR}(\tau, \eta, L_m(j))} \end{aligned} \tag{9}$$

We have $V_m(G), V_m(X[j]) \models \eta(L_m(j))$ by the inductive hypothesis. Since η is an inductive assertion for the reflective abstraction $\text{REFLECT}_\Pi(\eta)$, it consecutes w.r.t the transition $\text{MIRROR}(\tau, \eta, L_m(j))$:

$$\text{post}(\eta(L_m(j)), \text{MIRROR}(\tau, \eta, L_m(j)))\ \models\ \eta(L_m(j))\,. \tag{10}$$

Combining (9) and (10) we obtain

$$\begin{aligned} V_{m+1}(G), V_{m+1}(X[j]) \models \text{post}(\eta(L_m(j)), \text{MIRROR}(\tau, \eta, L_m(j)))\ \models \\ \models \eta(L_m(j))\ \ (= \eta(L_{m+1}(j)))\,. \end{aligned}$$

This completes our induction proof. □

B Soundness of Reflective Abstraction Interpretation

Definition 8 (Structural Pre-Order). *A sequential transition system*

$$\Sigma\colon \langle X, Loc, Trs, \ell_0, \Theta\rangle$$

structurally refines

$$\Sigma'\colon \langle X', Loc', Trs', \ell_0', \Theta\rangle\,,$$

written $\Sigma \preceq \Sigma'$, iff the following conditions hold:

1. *the program variables of Σ and Σ' are the same (i.e., $X = X'$);*
2. *the locations of Σ and Σ' are the same (i.e., $Loc = Loc'$);*
3. *the initial locations are the same (i.e., $\ell_0 = \ell_0'$);*
4. *the initial condition of Σ' over-approximates the initial condition of Σ (i.e., $\Theta \models \Theta'$); and*

5. *for any transition* $\tau'\colon \langle \ell'_{\text{src}}, \ell'_{\text{tgt}}, \rho' \rangle \in \mathit{Trs}'$, *there is a transition*

$$\tau\colon \langle \ell_{\text{src}}, \ell_{\text{tgt}}, \rho \rangle \in \mathit{Trs}$$

such that the pre- and post-locations are the same and the transition relation of τ' *over-approximates the transition relation of* τ *(i.e.,* $\ell_{\text{src}} = \ell'_{\text{src}}$, $\ell_{\text{tgt}} = \ell'_{\text{tgt}}$, *and* $\rho \models \rho'$*).*

It is clear that if $\Sigma \preceq \Sigma'$, then the behaviors of Σ' over-approximate the behaviors Σ.

Lemma 1 (Monotonicity of $\mathscr{F}$). *The concrete collecting semantics functional* $\mathscr{F}(\eta, \Sigma)$ *is monotone over (a) its first argument ranging over the lattice of concrete assertion maps ordered by* $\models$ *location-wise; and (b) its second argument* Σ *using the structural pre-order between sequential transition systems.*

Lemma 2 (Monotonicity of $\widehat{\mathscr{G}}_{\text{LAZY},\Pi}$). *The operator* $\widehat{\mathscr{G}}_{\text{LAZY},\Pi}(\widehat{\eta})$

$$\widehat{\mathscr{G}}_{\text{LAZY},\Pi}(\widehat{\eta}) \quad \stackrel{\text{def}}{=} \quad \mathsf{lfp}\ \widehat{\mathscr{F}}(\bot, \textsc{Reflect}_{\Pi}(\gamma \circ \widehat{\eta}))$$

is monotone over abstract assertion maps on to the abstract domain $\mathscr{A}$.

Corollary 1 (Lazy Reflective Abstract Interpretation Soundness). *Let*

$$\widehat{\eta}^* = \mathsf{lfp}\ \widehat{\mathscr{G}}_{\text{LAZY},\Pi}(\bot) \qquad \textit{and let map } \eta_{\mathit{inv}} \stackrel{\text{def}}{=} \gamma \circ \widehat{\eta}^*.$$

Then, for each location ℓ, $\eta_{\mathit{inv}}(\ell)$ *is a* 1*-index invariant of the system* Π.

Soundness of reflective abstract interpretation follows from the soundness of abstract interpretation and reflection soundness (Theorem 1).

C Empirical Evaluation: Benchmarks

Table 3 presents the benchmarks used in the empirical evaluation in Sect. 5 and summarizes the essential details of them. In this section, we describe them in detail for reference. Tbar is a simple parametrized barrier synchronization algorithm (Fig. 4). Wsteal is a parametrized array processing program, where each thread process a chuck of elements in a global array (Fig. 5). Cbar is a centralized barrier synchronization program,

Table 3. Benchmarks used in the empirical evaluation (Sect. 5). Legend: **Locs**: # locations, **Vars**: # shared + local vars, **Trans**: # transitions, **Props**: # properties to be verified.

ID	Locs	Vars	Trans	Props	Description
Tbar	3	6	4	4	Parametrized barrier synchronization
Wsteal	4	10	5	5	Parametrized workstealing over arrays
Cbar	7	13	11	9	Centralized parametrized barrier sync
Phil	10	14	14	14	Dining philosophers
Rb(m,n)		$mn+8$	$3mn+1$	mn	Swarming robots in a $m \times n$ grid

```
global count : int where count = N

thread P {
1   count := count - 1;
2   await (count = 0);
3 }
```

	Property	Description
basic	$N_1 + N_2 + N_3 = N$	All threads are at some position
bound	$count = N_1$	*count* equals number of thread at position 1
l_bound	$0 \leq count$	*count* is positive
u_bound	$count \leq N$	*count* is not greater than total number of threads

Fig. 4. Tbar: A simple synchronization barrier.

which allows reutilization of the barrier (Fig. 6). Phil is a variation of the classic dinning philosophers problem, with an unbounded number of processes acting as philosophers, but with a bounded number of resources (chopsticks) that the philosophers use (Fig. 7). Finally, Rb describes the behavior of a swarm of robots moving over a bounded grid (Fig. 8). In all cases, we use N to denote the number of threads in an instance of the parametrized system. Similarly, we use N_i to denote the number of threads currently at location i.

C.1 Tbar: A trivial barrier synchronization algorithm

The first example is a simple barrier synchronization algorithm, shown in Fig. 4. The global variable *count* is initialized with the total number of threads. Each thread decrements the global variable. No thread is allowed to reach line 3 until all threads have decremented the variable. The table at the bottom of Fig. 4 list the invariant properties that form the specification of the system.

C.2 Wsteal: A workstealing algorithm

The program Wsteal has already been introduced in the paper as motivating example, in Fig. 2. Threads access an array concurrently and perform some work over each element of the array independently. For the sake of simplicity, the specific array accesses has been removed form the example code. Fig. 5 shows the program source code and the specification with the invariant properties that we would like to prove for this program.

C.3 Cbar: A centralized barrier synchronization algorithm

Cbar, like Tbar, implements a memory barrier. The memory barrier implemented by Cbar can be reused inside a loop thanks to the use of the *sense* global variable. This program is shown in Fig. 6, alongside its specification.

```
global len  : int where len >= 0 ;
global next : int where next = 0;

thread P {
  local c   : int where c = 0;
  local end : int where end = 0;
1 atomic if (next + 10 <= len) { c := next; next := end := next + 10; }
2 while (c < end) {
3   c := c + 1;
  }
4 }
```

	Property	Description
basic	$N_1+N_2+N_3+N_4=N$	All threads are at some position
c_l_bound	$0 \leq c$	c is positive
c_u_bound	$c \leq len$	c does not go beyond *len*
next_bound	$0 \leq next$	*next* is positive
end_bound	$end \leq len$	*end* is bounded by *len*

Fig. 5. Wsteal: A parametrized array processing program.

C.4 Phil: Dining philosophers with bounded number of resources

Fig. 7 shows a variation of dining philosophers, inspired by [34] in which an arbitrary number of philosophers can be eating at each table position simultaneously, with a bounded number of resources. The specification appears in Table 4.

C.5 Rb: Robot Swarm

The last example consists of a swarm of robots moving across a $m \times n$ grid. The example is inspired in part by the work of Correll and Martinoli for modeling swarm robots that perform inspections over tight spaces such as the wing of an airplane [12].

Fig. 8 shows an instance of this program in a 2×2 board. The robots communicate with a central process that maps the number of robots in any grid cell. Each robot updates this central process with its position. Likewise, the central process updates each robot with a global map consisting of the number of robots in each cell. The details of this process are captured in our model by a shared memory view of the number of robots in a given cell.

If a given cell has more than a minimal number of robots, some of the robots in the cell may optionally migrate to a neighboring cell. This cell is chosen non-deterministically and the robot's velocity vector is set to point towards the cell. The robot then subtracts itself from the number of robots in the cell and moves for 10 time units in the specified direction. After the time units elapse, the robot re-evaluates its position and adds itself to the count of the number of robots in whatever cell it finds itself in.

```
global sense : int where sense = 0;
global count : int where count = N;

thread P {
  local localCount : int where localCount = N;
  local localSense : int where localSense = 0;
1 atomic if (localSense != 0) { localSense := 0; } else { localSense := 1 }
2 atomic localCount := count; count := count -1;
3 if (localCount > 1) {
4   await (sense = localSense);
  } else {
5   count := N;
6   sense := localSense;
  }
7 }
```

	Property	Description
basic	$N = \sum_{i=1}^{7} N_i$	All threads are in some location
sense_l_bound	$0 \leq sense$	*sense* is positive
sense_u_bound	$sense \leq 1$	*sense* is not greater than 1
localsense_l_bound	$0 \leq localSense$	*localsense* is positive
localsense_u_bound	$localSense \leq 1$	*localsense* is not greater than 1
count_l_bound	$0 \leq count$	*count is positive*
count_u_bound	$count \leq N$	*count* is not greater than N
localcount_l_bound	$0 \leq localCount$	*localCount* is positive
localcount_u_bound	$localCount \leq N$	*localCount* is not greater than N

Fig. 6. Cbar: A centralized synchronization barrier.

```
global cs1 : int where cs1 = 2;
global cs2 : int where cs2 = N;

thread P {
1 nondet_choice {
2   atomic await (cs2 > 1); cs2 := cs2 - 1;
3   atomic await (cs1 > 0); cs1 := cs1 - 1;
4   cs1 := cs1 + 1;
5   cs2 := cs2 + 1;
  } or {
6   atomic await (cs1 > 0); cs1 := cs1 - 1;
7   atomic await (cs2 > 0); cs2 := cs2 - 1;
8   cs2 := cs2 + 1;
9   cs1 := cs1 + 1;
  }
10 }
```

Fig. 7. Phil: Dining philosophers with a bounded number of resources.

Table 4. Analyzed properties for Phil (Fig. 7).

	basic
Prop.	$N = \sum_{i=1}^{10} N_i$
Desc.	All threads are in some location
	cons1
Prop.	$2 = cs1 + (N_7 + N_8 + N_9) + N_4$
Desc.	Resource 1 quantity (available/used) stays constant
	cons2
Prop.	$2 = cs2 + (N_3 + N_4 + N_5) + N8$
Desc.	Resource 2 quantity (available/used) stays constant
	res1
Prop.	$cs1 \geq 0$
Desc.	Resource 1 quantity is positive
	res2
Prop.	$cs2 \geq 0$
Desc.	Resource 2 quantity is positive
	use1
Prop.	$cs1 = 0 \Rightarrow (N_7 + N_8 + N_9) + N_4 \geq 1$
Desc.	If no resource 1 available, then some thread is using it
	use2
Prop.	$cs2 = 0 \Rightarrow (N_3 + N_4 + N_5) + N_8 \geq 1$
Desc.	If no resource 2 available, then some thread is using it
	actLim_res1
Prop.	$(N_7 + N_8 + N_9) + N_4 \leq 2$
Desc.	Number of threads using resource 1 is bounded by number of resources
	actLim_res2
Prop.	$(N_3 + N_4 + N_5) + N_8 \leq 2$
Desc.	Number of threads using resource 2 is bounded by number of resources
	actLim_res0
Prop.	$(N_3 + N_4 + N_5) \leq 1$
Desc.	Only one thread can own the last resource
	someProgress2_1
Prop.	$(N_2 > 0 \wedge cs2 \leq 1) \Rightarrow (N_3 + N_4 + N_5) + N_8 \geq 1$
Desc.	If resource 2 is required but unavailable, then some thread owns one and will release it
	someProgress2_0
Prop.	$(cs2 = 0) \Rightarrow N_8 \geq 1$
Desc.	If no resource 2 is available, then some thread owns one and will release it
	someProgress1_1
Prop.	$(N_3 > 0 \wedge cs1 = 0) \Rightarrow (N_7 + N_8 + N_9) + N_4 \geq 1$
Desc.	If resource 1 is required but available, then some thread owns one and will release it
	someProgress1_0
Prop.	$(N_6 > 0 \wedge cs1 = 0) \Rightarrow (N_7 + N_8 + N_9) + N_4 \geq 1$
Desc.	If no resource 1 is available, then some thread owns one and will release it

```
global x11 : int where x11 = N;
global x12 : int where x12 = 0;
global x21 : int where x21 = 0;
global x22 : int where x22 = 0;

thread P {
  local x  : int;
  local y  : int;
  local vx : int where vx = 0;
  local vy : int where vy = 0;
  local j  : int where j = 0
1   await ( x >= 0 && x <= 10 && y >= 0 &&
2             y <= 10 && N >= 6);
3   while (1 = 1) {
4     nondet_choice {
5       atomic await ( x >= 0 && x <= 10 && y >= 0
6                      && y <= 10 && x1y1 >= 2);
                vx := 1; vy := 0; x1y1 := x1y1 - 1; } or {
7       atomic await ( x >= 0 && x <= 10 && y >= 0
8                      && y <= 10 && x1y1 >= 2);
                   vx := 0; vy := 1; x1y1 := x1y1 - 1;     } or {
9       atomic await ( x >= 0 && x <= 10 && y >= 10 && y <= 20 && x1y2 >= 2);
                vx := 1; vy := 0; x1y2 := x1y2 - 1;     } or {
10      atomic await ( x >= 0 && x <= 10 && y >= 10 && y <= 20 && x1y2 >= 2);
                vx := 0; vy := -1; x1y2 := x1y2 - 1;     } or {
11      atomic await ( x >= 10 && x <= 20 && y >= 0 && y <= 10 && x2y1 >= 2);
                vx := 0; vy := 1; x2y1 := x2y1 - 1;     } or {
12      atomic await ( x >= 10 && x <= 20 && y >= 0 && y <= 10 && x2y1 >= 2);
                vx := -1; vy := 0; x2y1 := x2y1 - 1;     } or {
13      atomic await ( x >= 10 && x <= 20 && y >= 10 && y <= 20 && x2y2 >= 2);
                vx := -1; vy := 0; x2y2 := x2y2 - 1;     } or {
14      atomic await ( x >= 10 && x <= 20 && y >= 10 && y <= 20 && x2y2 >= 2);
                vx := 0; vy := -1; x2y2 := x2y2 - 1;
    }
15    j := 0;
16    while (j <= 9) {
17      x := x + vx;
18      y := y + vy;
19      j := j + 1;
    }
20    nondet_choice {
21      atomic await ( x >= 0 && x <= 10 && y >= 0 && y <= 10 ); x1y1 := x1y1 + 1;
    } or {
22      atomic await ( x >= 0 && x <= 10 && y >= 10 && y <= 20 ); x1y2 := x1y2 + 1;
    } or {
23      atomic await ( x >= 10 && x <= 20 && y >= 0 && y <= 10 ); x2y1 := x2y1 + 1;
    } or {
24      atomic await ( x >= 10 && x <= 20 && y >= 10 && y <= 20 ); x2y2 := x2y2 + 1;
    }
  }
25 }
```

Fig. 8. Rb(2,2): A set of robots moving around a 2×2 grid.

Table 5. Analyzed properties for Rb(2,2) (Fig. 8).

	Basic(i,j) $1 \leq i \leq 2, 1 \leq j \leq 2$
Prop.	$x_i y_j \geq 0$
Desc.	The number of robots in position i,j should never be negative.
	Velocity-Bounds
Prop.	$-1 \leq v_x \leq 1$
	$-1 \leq v_y \leq 1$
	$-1 \leq v_x + v_y \leq 1$
Desc.	Velocity of the robots must stay within bounds.
	xy Bounds
Prop.	$0 \leq x \leq 20$
	$0 \leq y \leq 20$
Desc.	(x,y) co-ordinates of robots must be in the grid.

D Detailed Location-wise Comparison of Invariants

Table 6 gives a detailed location-wise comparison of the strength of the various iteration schemes. This table is summarized earlier in Sect. 5, Table 2.

Table 6. Location-wise comparison of the strength of the various schemes for invariant inference. Legend: **L**: lazy, **E:** Eager, **E+:** Eager+, **I**: Inteference. In a comparison $A : B$, $+$ indicates that A obtained a stronger invariant at a location, $-$ means A obtained a strictly weaker invariant, $=$ means A and B obtained equivalent invariants, and $\neq$ means A and B obtained incomparable invariants. The comparison shows outcomes and number of locations (superscript).

ID	Dom	L:E	L:E+	L:In	E:In	E+:In	E:E+
Tbar	I	$-^{3}$	$-^{3}$	$+^{3}$	$+^{3}$	$+^{3}$	$=^{3}$
	P	$=^{3}$	$=^{3}$	$+^{1}=^{2}$	$+^{1}=^{2}$	$+^{1}=^{2}$	$=^{3}$
	O	$=^{3}$	$=^{3}$	$+^{1}=^{2}$	$+^{1}=^{2}$	$+^{1}=^{2}$	$=^{3}$
Wsteal	I	$+^{4}$	$+^{4}$	$+^{4}$	$+^{4}$	$+^{4}$	$=^{4}$
	P	$+^{4}$	$+^{1}\neq^{3}$	$=^{1}\neq^{3}$	$-^{2}\neq^{2}$	$+^{2}-^{1}=^{1}$	$-^{4}$
	O	$=^{4}$	$=^{4}$	$+^{2}=^{2}$	$+^{2}=^{2}$	$+^{2}=^{2}$	$=^{4}$
Cbar	I	$+^{6}\neq^{1}$	$+^{6}\neq^{1}$	$+^{7}$	$+^{7}$	$+^{7}$	$=^{7}$
	P	TO	TO	TO	$\neq^{7}$	$+^{1}\neq^{6}$	$-^{7}$
	O	TO	TO	TO	$+^{7}$	$+^{7}$	$=^{7}$
Phil	I	$+^{1}=^{9}$	$+^{1}=^{9}$	$+^{10}$	$+^{10}$	$+^{10}$	$=^{10}$
	P	$+^{10}$	$+^{10}$	$+^{1}=^{9}$	$-^{10}$	$-^{10}$	$=^{10}$
	O	TO	TO	TO	$+^{1}=^{9}$	$+^{1}=^{9}$	$=^{10}$
Rb(2,2)	I	$+^{22}$	$+^{22}$	$+^{22}$	$+^{22}$	$+^{22}$	$=^{22}$
	P	TO	TO	TO	$+^{2}-^{3}=^{17}$	$+^{2}-^{1}=^{19}$	$-^{2}=^{20}$
	O	TO	TO	TO	$+^{1}-^{4}=^{16}\neq^{1}$	$+^{2}=^{20}$	$-^{5}=^{17}$
Rb(2,3)	I	$+^{30}$	$+^{30}$	$+^{30}$	$+^{30}$	$+^{30}$	$=^{30}$
	P	TO	TO	TO	$+^{2}-^{3}=^{25}$	$+^{2}-^{1}=^{27}$	$-^{2}=^{28}$
	O	TO	TO	TO	$+^{1}-^{7}=^{21}\neq^{1}$	$+^{2}=^{28}$	$-^{8}=^{22}$
Rb(3,3)	I	$+^{43}$	$+^{43}$	$+^{43}$	$+^{43}$	$+^{43}$	$=^{43}$
	P	TO	TO	TO	$+^{2}-^{3}=^{38}$	$+^{2}-^{1}=^{40}$	$-^{2}=^{41}$
	O	TO	TO	TO	$+^{1}-^{10}=^{31}\neq^{1}$	$+^{2}=^{41}$	$-^{11}=^{32}$
Rb(4,4)	I	TO	TO	TO	$+^{74}$	$+^{74}$	$=^{74}$
	P	TO	TO	TO	$+^{3}-^{3}=^{68}$	$+^{3}-^{1}=^{70}$	$-^{2}=^{72}$
	O	TO	TO	TO	TO	TO	TO